PARSING FLOWCHARTS AND SERIES-PARALLEL GRAPHS

by

Jacobo Valdes

STAN-CS-78-682
DECEMBER 1978

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

Parsing Flowcharts and Series-Parallel Graphs

Jacobo Valdes

Computer Science Department
Stanford University
Stanford, California 94305

Abstract.

The main results presented in this work are an algorithm for the recognition of General Series Parallel (GSP) digraphs and an approach to the structural analysis of the control flow graphs of programs.

The GSP recognition algorithm determines in $O(n+m)$ steps whether an acyclic digraph with n vertices and m edges is GSP, and if it is, describes its structure in terms of two simple operations on digraphs. The algorithm is based on the relationship between GSP digraphs and the more standard class of TTSP multidigraphs.

Our approach to the analysis of flow graphs uses the triconnected components algorithm to find single-entry, single-exit regions. Under certain conditions -- that we identify -- this method will produce structural information suitable for the global flow analysis of control flow graphs in time proportional to the number of vertices and edges of the graph being analyzed.

This research was supported in part by National Science Foundation grant MCS75-22870 A02 and by Office of Naval Research contract N00014-76-C-0688. Reproduction in whole or in part is permitted for any purpose of the United States government.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
(14) REPORT NUMBER: STAN-CS-78-682 (9) Technical Rept.	3. RECIPIENT'S CATALOG NUMBER
(6) TYPE (and Subtitle): PARSING FLOWCHARTS AND SERIES-PARALLEL GRAPHS	5. TYPE OF REPORT & PERIOD COVERED: Technical, November 1978
	6. PERFORMING ORG. REPORT NUMBER: STAN-CS-78-682
7. AUTHOR(s): (10) Jacobo Valdes	8. CONTRACT OR GRANT NUMBER(s): (15) N00014-76-C-0688; NSF-MCS75-22870
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Computer Science Dep[illegible] Stanford University Stanford, California	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Office of Naval Research Department of the Navy Arlington, Va. 22217	(11) 12. REPORT DATE: Nov[illegible]78; 13. NUMBER OF PAGES: 233
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): ONR Representative: Philip Surra Durrand Aeromautics Bldg., Rm. 165 Stanford University Stanford, Ca. 94305 (12) 235 p.	15. SECURITY CLASS. (of this report): Unclassified; 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report): Releasable without limitations on dissemination.	
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)	
18. SUPPLEMENTARY NOTES	
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)	
20. ABSTRACT (Continue on reverse side if necessary and identify by block number): (see reverse side)	

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

$94 120

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

JB

The main results presented in this work are an algorithm for the recognition of General Series Parallel (GSP) digraphs and an approach to the structural analysis of the control flow graphs of programs.

The GSP recognition algorithm determines in $O(n+m)$ steps whether an acyclic digraph with n vertices and m edges is GSP, and if it is, describes its structure in terms of two simple operations on digraphs. The algorithm is based on the relationship between GSP digraphs and the more standard class of TTSP multidigraphs.

Our approach to the analysis of flow graphs uses the triconnected components algorithm to find single-entry, single-exit regions. Under certain conditions -- that we identify -- this method will produce structural information suitable for the global flow analysis of control flow graphs in time proportional to the number of vertices and edges of the graph being analyzed.

Acknowledgments

I seriously doubt that I would have finished my Ph.D. had it not been for Bob Tarjan who delivered me from the "n-th year blahs" by suggesting the topic of this thesis and supporting and encouraging me through its genesis. For this and for being generous with his time when I needed it I am deeply indebted to him.

The help of Gene Lawler and Andy Yao, who consented to read this work, is also acknowledged.

A few other people contributed to this effort in ways that are just as important but very much harder to describe. Phyllis Winkler, who was a wonderful mixture of friend, super secretary, and hatchet woman, and without whose help this work would have been still in progress. Luis Trabb Pardo (hermano!) who listened patiently to many hours of half-baked ideas -- without retaliating -- and remained a friend after I stole from him the summer of 1978. And Barbara Villalonga who was -- and is -- generous beyond reason.

Finally there is the proverbial "cast of thousands" who helped make my long stay at Stanford a very special period of my life. Among them: Rogelio, Anne, Jaime, Marga, Rob, Miris, "Sweet" Sue Graham, the "cocoteros", and the Bachacs. To all of them and to the many unnamed ones as well a very sincere thank you.

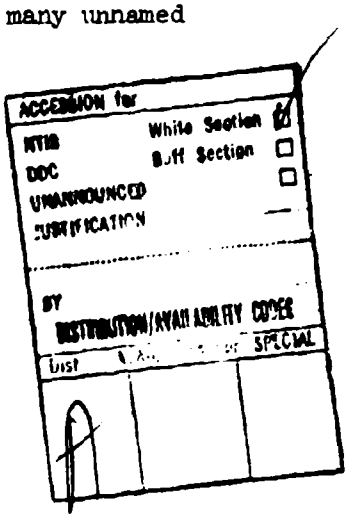


Table of Contents

Chapter 1. Introduction.

> parse (pärs) v. parsed, parsing. To describe the form, function and syntactical relationship of each part of a sentence [(L "pars", part].

Ever since its discovery, the theory of graphs has been a striking example of abstract mathematics originating from seemingly simple problems of the real world. Most of the classical problems in the field are such that they "... can be explained in five minutes by any mathematician to the so-called man on the street. At the end of the explanation, both will understand the problem, but neither will be able to solve." This description may not be quite fair to the mathematician (in fact Harary applied it to the four color problem that the mathematician has learned how to solve since the sentence was written) but is basically accurate.

The close relationship between these problems and everyday situations contrasts very sharply with the nature of the results provided by the classical theory of graphs. A good example of this contrast is Kuratowskii's characterization of planar graphs. Kuratowskii found a very simple condition (and one that can be described to a layman in simple terms) that a graph satisfies if and only if it can be drawn on the plane so that no two of its edges cross. This beautiful theorem helps very little however when trying to decide whether a particular graph can be drawn on the plane without crossings: Kuratowskii's proof gives no clues as to how to test a graph to decide whether it satisfies the condition.

The theory of graphs found applications in just about every branch of the sciences. The ubiquity of binary relations combined with the fact that binary relations are naturally represented as graphs and the intuitive

appeal of the diagrammatic representation of graphs helped to widen the field of applications of graph theory.

Many of the fields in which the theory of graphs found applications were eminently practical and found little use for non-constructive theorems. To a specialist in operations research a theorem that stated that there exists an optimal solution for any instance of a class of transportation problems only begged the question of how this solution could be computed. Thus with the applications came a shift in emphasis: it became important to know how to obtain solutions to certain problems.

This emphasis was accentuated with the widespread use of the digital computer and at the same time what was needed became more precise: a description of how to solve a problem had to be an algorithm suitable to be implemented in a digital computer.

It is in this context that this thesis should be considered. We are interested in designing efficient algorithms to solve graph theoretical problems arising from practical problems.

The qualification "efficient" is central to our concerns due to the intended practical applications of the algorithms we design. Most of the problems that we will consider can be solved by algorithms that are very simple both to describe and to implement. Unfortunately these simple algorithms use enormous amounts of computing resources (time and memory) and are therefore not very practical. Our task will be to decrease these requirements of computing resources usually at the expense of the simplicity of our algorithms.

The most obvious way of reducing the computing resources required by an algorithm is to implement it carefully. Sometimes the use of complex data structures and the careful design of its flow of control can result in considerable gains in the performance of an algorithm. In most cases, though, these gains are negligible -- a constant factor in most cases -- compared with what can be done by a mathematical analysis of the task that our algorithm has to perform. By discovering relations between the input and the desired output that are not immediately apparent, or by showing one task to be equivalent to another for which an efficient algorithm is known, one very often can realize enormous improvements which are well beyond what careful coding can achieve. For this reason, even though we are ultimately concerned with designing algorithms, we will spend almost all of our energies in the mathematical analysis of the problems to be solved.

In our discussion so far we have skipped over the crucial question of how the efficiency of an algorithm is to be measured. For this purpose we need (i) an abstract model of the machine in which our algorithms will be implemented, (ii) a resource whose utilization we want to minimize, and (iii) a consistent method of measuring the use of this resource by the abstract machine when running different algorithms that perform the same task. Our choices are the standard ones (see Appendix B and Chapter 1 of [AHO 76]): as an abstract machine we use a Random Access Machine (RAM), the resource we will measure is the number of steps that the RAM needs to solve a problem (which can be directly translated to the amount of processing time for a real machine), and we will measure it by associating

a size with the input and considering the number of steps taken by the machine as a function of the size of the input (see Appendix B). The RAM is chosen because it provides a realistic model of most present day digital computers. The processing time is selected as the resource to be minimized because until very recently was the most expensive resource; even though this is not the case today in some applications (due to the microprocessor), it still is the limiting resource in most cases and the amount of time used provides a bound on the utilization of many other resources -- like memory.

It is worth noting that even though it might appear that the tremendous increase in the speed of digital computers would decrease the importance of efficient algorithms, the opposite is true. The discussion given in Chapter 1 of [AHO 76] is very illuminating in this respect.

In the paragraphs that follow we give a general description of the problems considered in this thesis. In this description we use for the first time some technical terms, most of them standard graph theoretical terminology. The reader not familiar with the terms used can find their definition in Appendix A.

The algorithms that we will present in the rest of this work fall into two basic classes: recognition algorithms and parsing algorithms.

We say that Algorithm A recognizes a class of graphs C, if A answers "yes" when given as input a member of C, and answers "no" when given a graph that is not a member of C.

The verb "to parse" is commonly used in the Computer Sciences literature in a way that stretches somewhat its standard meaning. By parsing a graph we mean its analysis in terms of the interrelationships between some of its subgraphs and our parsing algorithms will perform this analysis.

The distinction between these two classes of algorithms is not quite as clear as the previous two paragraphs may lead one to believe. Some of the algorithms that we will present recognize classes of graphs by attempting to parse their inputs, knowing that they will succeed if and only if the input belongs to the class to be recognized.

The original contributions of our work are two: an algorithm to recognize and parse a class of directed graphs called General Series Parallel, and the use of the decomposition of a multigraph into triconnected components to analyze the flow of control of programs. In the following paragraphs we will describe in some more detail these problems and their applications.

General Series Parallel are directed acyclic graphs whose transitive closures form a class with a simple recursive definition. Their applications are related to problems of scheduling under constraints. In this application of directed graphs, vertices represent tasks to be executed by a processor (or processors) and edges represent constraints on how these tasks may be executed, so that if there is an edge going from vertex x to vertex y, task x has to be completed before task y is started. The problem in general is to find an order of execution of the tasks that satisfies the constraints and that minimizes some function of the tasks (like the total elapsed time needed to complete all the jobs). There are endless variations of this basic schema according to what function is to be minimized, the

number, type and arrangement of the processors, etc. Many of these problems arise in practice, and a good number of them are NP-complete (see [AHO 76] for a definition of NP-completeness) when we want an optimal solution for arbitrary precedence constraints, which basically means that no efficient algorithm is likely to exist to solve these problems.

This situation makes partial solutions for these problems more interesting. One can relax the optimality condition and simply ask for a solution that is guaranteed to be close to the optimal, or one can restrict the types of constraints that are acceptable and try to find efficient algorithms to solve these simpler problems.

General Series Parallel digraphs turn out to be useful for the second approach described: there exists a whole class of NP-complete scheduling problems for arbitrary constraints that can be solved by efficient algorithms when the contraints form a General Series Parallel digraph. (See [LAW 78], [MOM 77], [LAW 77], [SID 76].) It is therefore interesting to decide in an efficient way whether a given set of constraints forms a General Series Parallel digraph so the efficient algorithm can be applied to the corresponding scheduling problem. In Chapter 5 we will present an algorithm that performs this recognition task in a number of steps proportional to the number of vertices and edges of the directed graph to be tested.

Aside from the main application just described, our recognition algorithm has a few other interesting properties. The algorithm exhibits an uses the relationship between General Series Parallel digraphs and the more standard class of Two Terminal Series Parallel multigraphs which has been extensively applied to model electrical circuits. (See [DUF 65],

[LAW 60], [RIO 72], [WEI 71], [WEI 75].) Furthermore, whenever the input of our algorithm is a General Series Parallel digraph, we will obtain not only a "yes" answer as output, but also a parse of the graph. This parse permits the solution of several problems for General Series Parallel graphs by algorithms that are more efficient than the best known algorithms to solve the same problems for arbitrary directed graphs. These problems include transitive closure, transitive reduction and a restricted version of graph isomorphism.

Our second contribution is related to the application of directed graphs to model the flow of control of programs. In this standard technique, the vertices of a directed graph represent a sequence of program operations that are always executed serially and the edges represent transfers of control between such sequences.

The flow of control of most programs can be accurately described in this form so graph theoretical results can be applied to the problem of analyzing the flow of control of programs. In most cases, the information that one wishes to obtain is the following:

(i) given a point in the program, find what has happened before control reaches that point,

(ii) given a point in the program, find what can happen after control leaves that point.

Being able to answer questions of this type is a big step towards a solution of many problems that arise in the design of compilers. It is particularly useful during the code generation or code optimization phases to solve problems like register allocation, common subexpression elimination, code motion, etc.

The "classical" approach to this problem ([ALL 70], [COC 70], [GRA 76], [HEC 72], [HEC 74], [HEC 77], [KEN 71]) restricts the type of digraphs to be studied to flowgraphs: directed graphs with one starting vertex (corresponding to the first executable statement of a program) from which all of the other vertices can be reached. The analysis of these graphs is then carried out in terms of intervals: subgraphs that have basically a flowgraph structure in that they have a single entry vertex. The most efficient way known of performing this analysis ([TAR 74]) is loosely based on the systematic simplification of the flowgraph to be studied using standard subgraph replacement rules.

Our approach will be slightly different. We will restrict ourselves to a type of directed graphs called hammocks: directed graphs with two distinguished vertices, one of them being an entry vertex and the other an exit vertex, such that every vertex can be reached from the entry and the exit can be reached from any vertex. Our analysis of such graphs will be performed in terms of subhammocks: subgraphs that have a hammock structure in having two distinguished boundary vertices, one of them an entry into the subgraph and the other one an exit. We will show how this analysis can be carried out in an efficient way using an algorithm due to Hopcroft and Tarjan ([HOP 73]) that breaks up a graph into triconnected pieces. We will discuss the problems involved in applying this technique to general hammocks and describe how the problems diminish or disappear when we restrict ourselves to special classes of hammocks like proper programs ([GAN 77]) or structured programs ([DAH 72]).

Although the domain of application of the two algorithms that we will present are very different, the way in which they help solve the problems

to which they are applied is just an instance of the common technique of "divide and conquer". By providing a description of the structure (a parse) of a graph, our algorithms allow to solve the scheduling or flow questions in a large graph G by solving similar problems in trivial subgraphs of G, and then "pasting" these solutions together to form the solution on G.

The way in which we will decompose a graph in the "divide" part of the "divide and conquer" strategy will be by finding separation pairs. In the general cases we will use Hopcroft and Tarjan's algorithm to break a graph into triconnected pieces, and in some particular cases we will use simpler methods based on replacement systems. In either case our algorithms will break up a graph into pieces that have relatively simple structures and that fit together in a natural way to form the original graph. The simple structure of the pieces will make the solution of the problems mentioned above on them an efficient process, and the natural way in which the pieces fit together will facilitate the construction of the total solution from the solutions for the pieces.

The results just described and some others of lesser importance are distributed over the next five chapters. In an effort to give the reader a better idea of the overall organization of this work before plunging into the details, we end this introduction by giving a chapter by chapter outline of the rest of this thesis.

In Chapter 2 we describe and study a tool commonly used to parse graphs: Replacement Systems. We review some known results on the subject and define some particular systems that we will use in later chapters.

Chapter 3 is a presentation of a collection of facts related to a class of graphs called Two Terminal networks. We review a theory of Two Terminal network decomposition, describe the triconnected components algorithm, show that the theory of breaking a graph into triconnected pieces is a constructive version of the theory of network decomposition, and introduce a subset of Two Terminal networks which has been extensively used to model electrical circuits called Two Terminal Series Parallel networks.

Chapter 4 is devoted to the study of Two Terminal Series Parallel multidigraphs which (as we may expect) are closely related to the Two Terminal Series Parallel networks introduced previously. The emphasis throughout this chapter is on the properties of these multidigraphs that will be used in the recognition of General Series Parallel digraphs.

Chapters 5 and 6 contain the main contributions of this work. Chapter 5 is a detailed description of the recognition procedure for General Series Parallel digraphs and its consequences, while Chapter 6 includes the application of the triconnected components algorithm to the analysis of the flow of control of programs.

Most of the material contained in this work can be grasped at an intuitive level, so an effort has been made to keep the presentation as clear and uncluttered as possible. As part of this effort, most of the proofs have been removed from the text and included as an appendix. We hope that in doing that we have made our main results more accessible to the casual reader.

Chapter 2. Replacement Systems.

In this section we will study Replacement Systems as tools to parse graphs. We will start our discussion by describing these systems and our intended application in an informal way, and then provide a more formal description which includes the proof of an important property of the particular replacement systems that we will use in later chapters.

Replacement Systems will be used as a convenient way of systematically simplifying a graph. Basically, our replacement systems consist of a collection of reduction rules which specify that certain subgraphs can be replaced by smaller graphs during the simplification process.

The systems that we will use are based on the following rules:

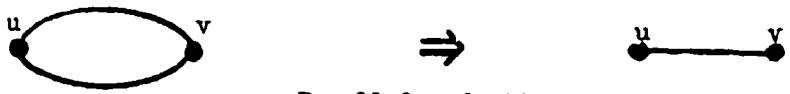
Series Reduction: replace two edges (u,v) , (v,w) in series (that is, such that v has degree two) by an edge (u,w) .

Parallel Reduction: replace two edges (u,v) , (u,v) in parallel by a single edge with the same endpoints.

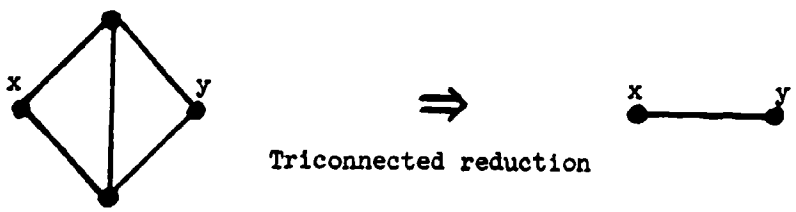
Triconnected Reduction: let $G_s = \langle V_s, E_s \rangle$ be the subgraph induced by V_s . If G_s has exactly two boundary vertices, x , y , contains at least four vertices and $G'_s = \langle V_s , E_s \cup \{(x,y)\} \rangle$ is triconnected, then replace G_s by an edge (x,y) .

All three of the rules specify subgraphs having exactly two boundary vertices to be replaced by a single edge joining the two boundary vertices. These three rules are illustrated in Figure 2.1. Replacement systems based on these three rules are common in the computer science and graph theory literature (see [WAL 78], [DUF 65], [HAR 72], [PRA 78] or [LIU 77]).

Series reduction



Parallel reduction



Triconnected reduction

Figure 2.1.

Consider the following process of simplification of a graph, G_0, using a set of reduction rules like the ones just described. First we identify the subgraphs of G_0 to which reductions can be applied, then we apply these reductions, one at a time, thus transforming G_0 into G_1. We repeat the process on G_1 to obtain a new graph G_2, repeat the process on G_2 once again to obtain G_3 etc., until we obtain an irreducible graph G_k, that is, a graph that cannot be simplified with our reduction rules. Consider two consecutive elements, G_i and G_{i+1}, of the sequence of graphs $G_0, G_1, \ldots, G_k$, just described. We can look at G_{i+1} as a simplified version of G_i, with the set of reductions, R, used to transform G_i into G_{i+1} being a description of the details that have been suppressed. Thus G_i can be represented by G_{i+1} and R.

Using this method one can describe a graph, G, by exhibiting the irreducible graph obtained from G by application of the reduction rules, and the sequence of reductions used to obtain it. In many situations this approach gives a useful and concise representation of the structure of a graph, and it is in this way that Replacement Systems will be used to parse graphs in the following chapters.

The simplification process presented above was described in a way that glossed over an important problem: the reductions applicable to a given graph may be mutually exclusive. As an example consider the graph of Figure 2.2 and assume that we are trying to simplify it using series reductions. Two series reductions can be applied to that graph: one involves edges 4 and 5, the other involves edges 5 and 6. These reductions are mutually exclusive in that once one of them is applied, one of the edges involved in the other is eliminated so the second cannot be applied any longer.

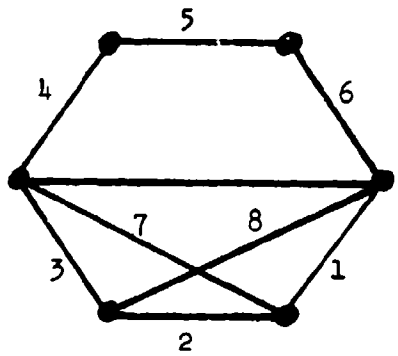


Figure 2.2.

Although it is clear that this matter is not very important in our example (by symmetry, if for no other reason), in a general case, selection of one or another of two mutually exclusive reductions may give rise to radically different descriptions of the graph. This possible multiplicity is undesirable in most practical applications, a fact that makes particularly useful reduction rules for which one can guarantee the following: given a graph G, the irreducible graph G_k obtained from G by repeated application of the reduction rules is unique and does not depend on the choices among mutually exclusive reductions. Replacement Systems for which this guarantee can be given are said to possess the Church - Rosser property, or to be Church - Rosser. All the Replacement Systems that we will use possess this property.

The remainder of this chapter contains a more precise description of some of the concepts already discussed. We provide formal definitions of replacement systems and the Church - Rosser property, define the replacement systems that we will use in later chapters and briefly review the literature relevant to proving the Church - Rosser property for these systems.

A binary relation, $\rightarrow$, on a set S, is a subset of $S \times S$. We will write $a \rightarrow b$ to indicate that the pair (a,b) belongs to the set $\rightarrow$. We will say that " b can be obtained from a " or that " a reduces to b " if $a \rightarrow b$, and call the operation of replacing a by b a "reduction".

The transitive reflexive closure, $\rightarrow^*$, of a binary relation, $\rightarrow$, defined on a set S, is the binary relation given by $a \rightarrow^* b$ if and only if $a = b$ or there exists a sequence of elements of S, $a_1 a_2 \ldots a_k$ such that $a_1 = a$, $a_k = b$ and $a_i \rightarrow a_{i+1}$ for $1 \leq i < k$.

An element, a, of a set S is <u>irreducible</u> under a binary relation, $\rightarrow$, defined on S if there is no element, b, of S such that $a \rightarrow b$. The <u>completion</u> of a binary relation, $\rightarrow$, defined on a set S, is the binary relation defined by $a \tilde{\rightarrow} b$ if and only if $a \rightarrow^* b$ and b is irreducible under $\rightarrow$.

A structure consisting of a set S and a binary relation, $\rightarrow$, on S is called a <u>replacement system</u>: $(S, \rightarrow)$.

A replacement system $(S, \rightarrow)$ is <u>finite</u> if for each element, a, of S there is a bound on the length of the longest sequence $a_1, a_2, \ldots, a_k$ such that $a_1 = a$ and $a_i \rightarrow a_{i+1}$ for $1 \leq i < k$. A replacement system $(S, \rightarrow)$ is finite <u>Church - Rosser</u> (FCR) if it is finite and $\tilde{\rightarrow}$ defines a function on S, that is, if $a \tilde{\rightarrow} b$ and $a \tilde{\rightarrow} c$ implies $b = c$.

Corresponding to the reduction rules defined earlier we can define three replacement systems on the set of all multigraphs:

<u>Series Replacement System</u>: $x \rightarrow_s y$ if the multigraph y can be obtained from the multigraph x by a single series reduction.

<u>Parallel Replacement System</u>: $x \rightarrow_p y$ if the multigraph y can be obtained from the multigraph x by a single parallel reduction.

<u>Triconnected Replacement System</u>: $x \rightarrow_t y$ if the multigraph y can be obtained from the multigraph x by a single triconnected reduction.

<u>Theorem 2.1</u>. The Series Replacement System, the Parallel Replacement System, and the Triconnected Replacement System are FCR.

<u>Proof</u>. See [WAL 78]. □

Two replacement systems that we will use in the following chapters are defined on the set of all multigraphs and their binary relations are best described as the union of some of the binary relations just defined:

Series Parallel Replacement System (SPRS): $\rightarrow_{sp} = \rightarrow_s \cup \rightarrow_p$.

Universal Replacement System (URS): $\rightarrow_u = \rightarrow_s \cup \rightarrow_p \cup \rightarrow_t$.

We will use still one more replacement system defined on the set of all multidigraphs, but otherwise identical to the SPRS. The definitions of Series Reduction and Parallel Reduction given at the beginning of the chapter can be interpreted as operations on multidgraphs due to the two possible interpretations of the terms "in series" and "in parallel" (see Appendix A). We can thus imagine the binary relations $\rightarrow_s$, $\rightarrow_p$, and $\rightarrow_{sp}$ as defined on the set of all multidigraphs and consider the Directed Series Parallel Replacement System (DSPRS) as defined by the set of all multidigraphs and the binary relation $\rightarrow_{sp}$.

The last three replacement systems defined have in common with many other useful replacement systems the property of being most naturally defined as the union of several simpler systems. The work of Rosen [ROS 73] and Sethi [SET 74] simplifies considerably the task of proving that a system of this type is FCR. We will review here two of their results that are useful in proving that the SPRS, the URS, and the DSPRS are FCR.

Let $\rightarrow_1$ and $\rightarrow_2$ be two binary relations on a set S . We say that $\rightarrow_1$ commutes with $\rightarrow_2$ if $a \rightarrow_1^* b$ and $a \rightarrow_2^* c$ implies that for some element $d \in S$, $b \rightarrow_2^* d$ and $c \rightarrow_1^* d$.

Theorem 2.2 [ROS 73]. Let $(S, \rightarrow)$ be a replacement system in which $\rightarrow = \bigcup_{1 \leq i \leq n} (\rightarrow_i)$ and let $(S, \rightarrow_i)$ be FCR for $i \in \{1,2,\ldots,n\}$.

If $\rightarrow_j$ commutes with $\rightarrow_k$ for $j,k \in \{1,2,\ldots,n\}$, then $(S, \rightarrow)$ is FCR. □

This theorem establishes the importance of the concept of commuting to prove the Church - Rosser property of composite replacement systems and the following lemma eases the task of proving that two binary relations on a set commute.

Lemma 2.1 [ROS 73], [SET 74]. Let $\rightarrow_1$ and $\rightarrow_2$ be two binary relations defined on a set S. If $a \rightarrow_1 b$ and $a \rightarrow_2 c$ implies that there exists an element $d \in S$ such that $b \rightarrow_2^* d$ and $c \rightarrow_1^* d$ then $\rightarrow_1$ commutes with $\rightarrow_2$. □

These results can be used to prove that the replacement systems that interest us are FCR as follows:

Lemma 2.2.

(a) $\rightarrow_s$ commutes with $\rightarrow_p$ (for directed or undirected multigraphs).

(b) $\rightarrow_s$ commutes with $\rightarrow_t$.

(c) $\rightarrow_p$ commutes with $\rightarrow_t$.

Proof. See discussion in [WAL 78]. □

The proof given by Walsh [WAL 78] of part (a) of Lemma 2.2 is for undirected multigraphs exclusively. His arguments can nevertheless be trivially modified to prove the proposition for directed multigraphs.

We can now state the following:

Theorem 2.3. The Series Parallel Replacement System (SPRS), Universal Replacement System (URS), and Directed Series Parallel Replacement System (DSPRS) are finite Church - Rosser.

Proof. Follows immediately from Theorems 2.1 and 2.2 and Lemma 2.2. □

Chapter 3. Two Terminal Networks.

3.1 Introduction.

In this chapter we study a class of undirected multigraphs called two terminal (TT) networks. These multigraphs have a wide variety of applications and a subset of them called two terminal series parallel (TTSP) networks have been extensively studied because of their applications to the design and analysis of electric circuits.

The goal of this chapter is to show the basic equivalence between the theory of TT network decomposition, the decomposition of a biconnected multigraph into triconnected components and the parsing of a graph using the Universal Replacement System. We will attempt to unify our discussion around the theory of triconnected decomposition because of its basic algorithmic flavor.

We will start the chapter by providing the basic definitions and reviewing a theory of TT network decomposition following the presentation of Walsh [WAL 78]. Walsh's main goal is to count certain classes of networks. As a result the theory that he presents is non-constructive and not well suited to the design of efficient algorithms to obtain the decompositions it postulates. In his work, Walsh appears to be merely reviewing a well established theory due to various Russian authors. Unfortunately most of the references that he provides have not been translated so it has not been possible to ascertain whether some algorithmic theory is developed in any of them, although it does not seem likely given their theoretical slant.

The theory of TT network decomposition is almost equivalent to the theory of breaking a multigraph into triconnected components developed by

Whitney [WHI 32] and Tutte [TUT 66]. These theories are again non-algorithmic, but more recently Hopcroft and Tarjan [HOP 73] have given an efficient algorithm to perform the decomposition of a biconnected multigraph into triconnected pieces. We will review the basic aspects of this algorithm and show how it can be used to compute the decomposition described by the main theorem of the theory of TT network decomposition.

We will then describe the connections between the decomposition given by the triconnected components algorithm and the parsing of a TT network using the Universal Replacement System described in the previous chapter. Although reduction systems are not particularly useful to parse TT networks, some of the features that make this parsing method useful in other cases are more natually introduced in the context of this chapter.

Finally we will turn our attention to the class of Two Terminal Series Parallel networks. We will show how the basic theory can be extended in several important aspects and explain how the Series Parallel reduction system can be efficiently used to parse these networks. TTSP networks have been extensively studied, mostly as models for circuits. (See [DUF 65], [LAW 60], [RIO 42], [SCO 65], [WEI 71], [WEI 75].) Our presentation will try to unify several pr es of these networks and show how they can be derived from the general theory of TT network decomposition and the triconnected components algorithm.

3.2 Decomposition of Two Terminal Networks.

A two terminal (TT) network is an undirected multigraph in which exactly two vertices are distinguished. The distinguished vertices are called the terminals of the network, and all other vertices are said to be internal. We will assume that the terminals are joined by a distinguished edge that we will call the return edge.

A TT network with at least three edges -- including the return edge -- is nontrivial (opposite: trivial). A simple path $x \Rightarrow^* y$ in a TT network is a terminal path if x and y are the terminals. A TT network is firmly connected if for every internal vertex, v, there is a terminal path that includes v. Figure 3.1 shows several examples of TT networks; examples (a) and (b) are not firmly connected while (c) and (d) are.

There is a direct relationship between the concepts of firm connectivity and biconnectivity, given by the following lemma:

Lemma 3.1.

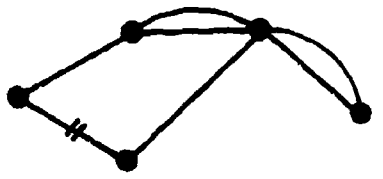
(a) A firmly connected TT network is biconnected.

(b) A biconnected multigraph with any two adjacent edges as terminals is a firmly connected TT network.

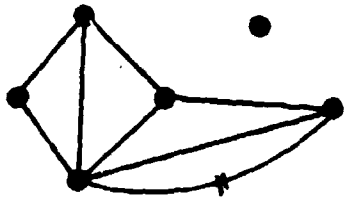
Proof. See [WAL 78]. □

Most TT networks that arise in practice are firmly connected. For this reason we will assume that all the multigraphs mentioned in this chapter are biconnected unless we explicitly state the opposite. The proof of Lemma 3.1 depends on the adjacency of the terminals of any TT network. This is the precise reason why the return edge was introduced, and even though all the results presented in this chapter can be reformulated without assuming its existence, our assumption simplifies some of the arguments considerably.

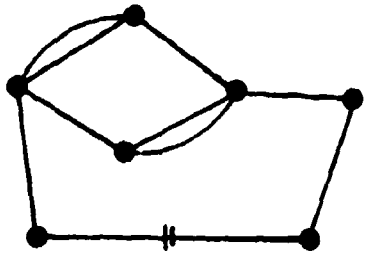
A subgraph of a TT network with exactly two boundary vertices that does not contain the return edge is a subnetwork. The two boundary vertices are the terminals of the subnetwork.



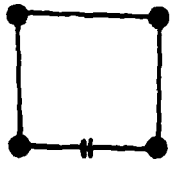
(a)



(b)



(c)



(d)

Figure 3.1. Distinguished edges of the TT networks are marked by a double bar.

Lemma 3.2. Every subnetwork of a firmly connected TT network is firmly connected.

Proof. (See Appendix C.) □

Let N_1 and N_2 be TT networks and let $e = (u,v)$ be an edge of N_2 distinct from its return edge. Consider the following operation (see Figure 3.2):

(i) Delete the return edge of N_1.

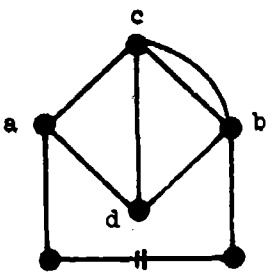
(ii) Replace (u,v) by the multigraph resulting from (i) by identifying one of the terminals of N_1 with u, and the other terminal with v.

We call this operation the replacement of $e = (u,v)$ by N_1.

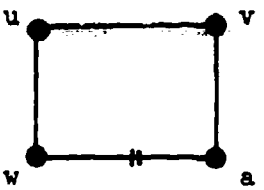
Let $N, N_0, N_1 \ldots N_k$ be TT networks such that N can be obtained by replacing some edges of N_0 by the networks $N_1, N_2 \ldots N_k$. If all the TT networks $N_0, N_1 \ldots N_k$ are non trivial and $k \geq 1$ we say that they form a decomposition of N with core N_0 and components $N_1, \ldots, N_k$. The condition that all the networks be non trivial is designed to exclude pseudo-decompositions in which either the core or some component is identical to the network being decomposed (see Figure 3.3). Note that because of the way replacement has been defined, the return edge of the core of a decomposition is the return edge of the TT network decomposed.

A TT network is indecomposible if it has no decomposition. Indecomposible TT networks are of one of three types:

Lemma 3.3. A nontrivial indecomposible TT network is either a triangle, a triple bond, or a triconnected graph with at least four vertices.



N_1



N_2

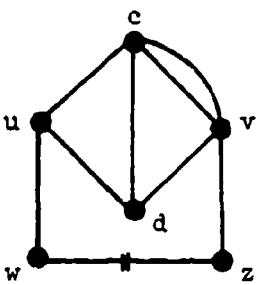
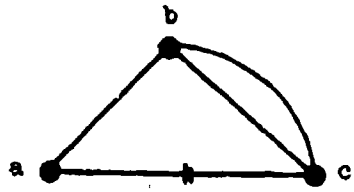
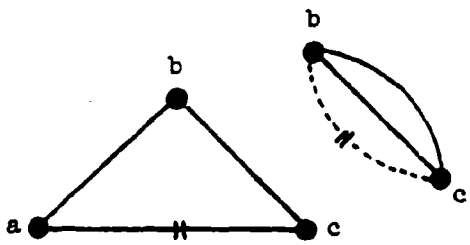


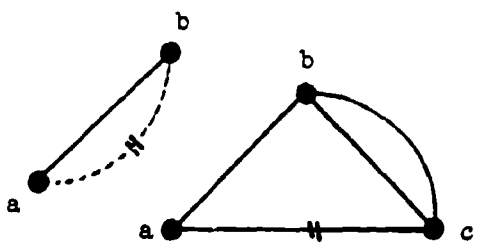
Figure 3.2.



N



A decomposition of N



A pseudo-decomposition of N

Figure 3.3.

Proof. (See Appendix C.) □

A decomposition whose core is a polygon (bond) is called a series decomposition (parallel decomposition). A decomposition whose core is a triconnected graph with at least four vertices is a t-decomposition. A TT network that has a series decomposition is an s-network, if it has a parallel decomposition it is a p-network, and if it has a t-decomposition it is a t-network.

A series decomposition is canonical if none of its components is an s-network, and a parallel decomposition is canonical if none of its components is a p-network.

Using these definitions we can state the basic theorem of the theory of TT network decomposition:

Theorem 3.1 [Trahktenbrot's theorem].

(a) A TT network is either indecomposible or is of exactly one of the types s , p , or t .

(b) An s-netowrk that is not a polygon has a unique canonical series decomposition.

(c) A p-network that is not a bond has a unique canonical parallel decomposition.

(d) A t-network has a unique t-decomposition. □

This formulation of Trahktenbrot's theorem is almost identical to the one given by Walsh ([WAL 78]) which also provides a proof for it.

The most important consequence of Trahktenbrot's theorem from our point of view is that it defines a unique way of breaking up a TT network into smaller networks that we will call Trahktenbrot's repeated decomposition:

-- Let N be a TT network. If N is a polygon, a bond or a triconnected graph with four or more vertices, N is not decomposed.

-- Otherwise N will be of exactly one of the types s, p, or t according to Theorem 3.1(a). In each case the theorem gives us a description of a <u>unique</u> way of decomposing $N = N_0, N_1, \ldots, N_k$ such that the core, N_0, is either a polygon, a bond or a triconnected graph with at least four vertices, and if N_0 is a polygon (bond) no component has a decomposition whose core is a polygon (bond).

-- We carry out this process repeatedly, breaking up the components $N_1 \ldots N_k$, then the components thus obtained and so on until no more networks can be decomposed.

-- In this manner we obtain from N a set of TT networks that satisfy:

(i) They are all polygons, bonds, or triconnected graphs with at least four vertices.

(ii) N can be constructed by appropriate replacement operations between the members of the set without ever replacing an edge of a polygon (bond) with a network of type s (p).

(iii) The set of TT networks is uniquely determined by N.

Trahktenbrot's theorem guarantees the existence of the set of TT networks just described, but does not provide information that leads directly to an efficient algorithm to compute it. We will now review the triconnected components algorithm of Hopcroft and Tarjan and show that when given a firmly connected TT network as input, it computes the set of TT networks described above.

3.3 The Triconnected Components Algorithm.

The algorithm that we will describe takes as input a biconnected multigraph G, and produces as output a set of polygons, bonds, and triconnected graphs that is unique. We will describe the functional relationship between the input and output of this algorithm without explaining the way the algorithm really works. (For details see [HOP 73].)

Consider the following operation on a biconnected multigraph $G = \langle V, E \rangle$.

(i) Find a separation pair a, b giving classes $E_1, E_2, \ldots, E_k$, then merge these classes into two disjoint sets E' and E'' each containing at least two edges (see Appendix A) with $E = E' \cup E''$.

(ii) Consider the multigraphs $G' = \langle V(E'), E' \cup \{(a,b)\} \rangle$ and $G'' = \langle V(E''), E'' \cup \{(a,b)\} \rangle$ where $V(E')$ stands for the vertices of G incident to edges in E' and $V(E'')$ for the vertices of G incident to E edges in E''.

This process is called splitting G, and G' and G'' are called the split graphs of G. The new edges -- (a,b) -- added to each of the split graphs are called virtual edges and they are assumed to be labelled so that they are identified with the split operation that creates them.

Suppose that a biconnected multigraph G is split, its split graphs are again split and so on until no more splits are possible. The graphs obtained in this way are called the split components of G. The split components of a biconnected multigraph are of one of three types: triangles, triple bonds, or triconnected graphs with four or more vertices. The set of split components of a biconnected multigraph is not unique as Figure 3.4 shows.

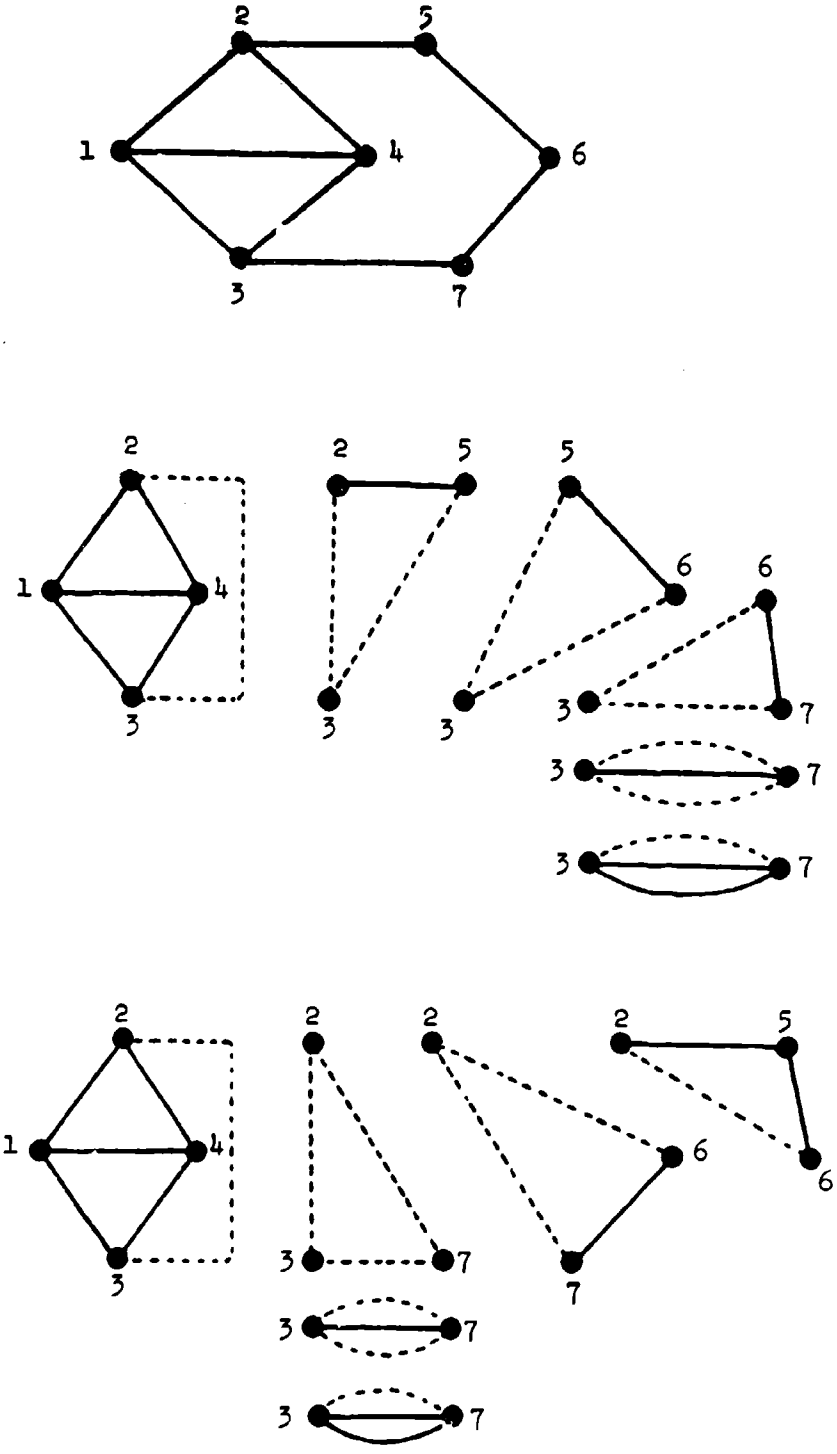


Figure 3.4. Two sets of split components of a biconnected multigraph.

Let $G' = \langle V', E' \rangle$ and $G'' = \langle V'', E'' \rangle$ be two split graphs of a biconnected multigraph sharing a virtual edge (a,b). That is, each graph contains an edge (a,b) and both edges were introduced by the same split operation. The operation of merging G' and G'' produces $G = \langle V_1 \cup V_2, E_1 - \{(a,b)\} \cup E_2 - \{(a,b)\} \rangle$. (This operation is identical to the replacement operation defined for TT networks except for the asymmetry introduced by our requirement that in a replacement one of the edges eliminated be the return edge of one of the two TT networks involved.)

Let us now consider the following process on a biconnected multigraph G:

(i) Split G repeatedly to obtain a set of triple bonds $\mathcal{B}_3$, a set of triangles $\mathcal{T}$, and a set of triconnected graphs with four or more vertices $\mathcal{L}$.

(ii) Merge the elements of $\mathcal{B}_3$ as much as possible to obtain a set $\mathcal{B}$ of bonds.

(iii) Merge the elements of $\mathcal{T}$ as much as possible to obtain a set of polygons $\mathcal{P}$.

The set $\mathcal{B} \cup \mathcal{P} \cup \mathcal{L}$ is the set of triconnected components of G. Figure 3.5 shows the triconnected components obtained by merging the split components of Figure 3.4; note that each edge of the original graph belongs to exactly one triconnected component and each virtual edge to exactly two components.

Theorem 3.2 [Hopcroft and Tarjan [HOP 73]]. The set of triconnected components of a biconnected multigraph is unique and can be computed in $O(n+m)$ steps for a multigraph with n vertices and m edges. □

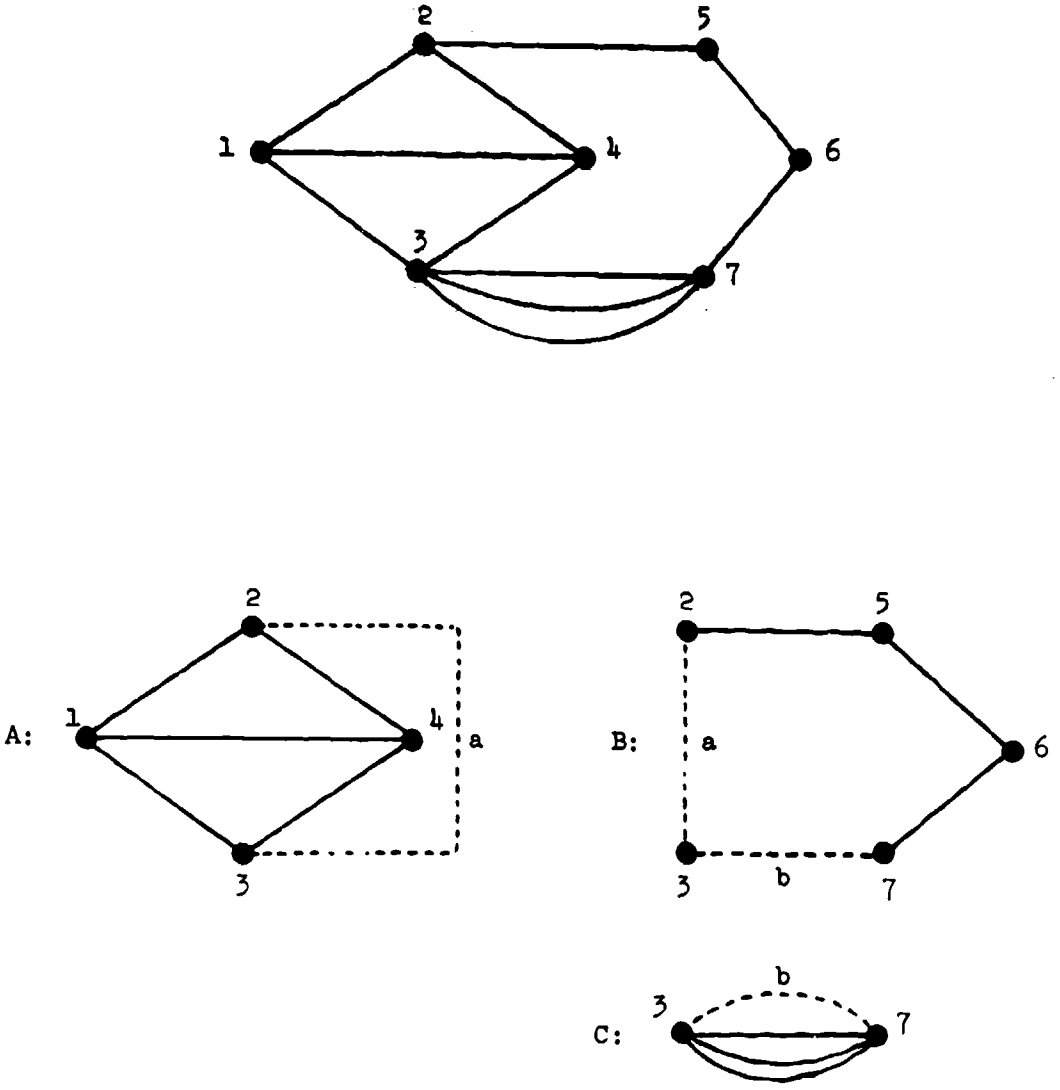


Figure 3.5.

In the next section we will discuss at length the relationship between the triconnected components algorithm and the Trahktenbrot repeated decomposition described in the previous section. The equivalence of these two theories can be grasped at an intuitive level by looking at the relationship between two pairs of operations: merging and replacement (which are basically identical) and splitting a biconnected graph and decomposing a TT network. The relationship between these last two operations can be made clear by the following lemma:

Lemma 3.4. Let N be a firmly connected TT network and N_1 a non trivial proper subnetwork of N with boundary vertices x and y.

(a) x, y is a separation pair of N.

(b) There is at least one decomposition of N in which N_1 is a component.

Proof. [See Appendix C.] □

The output of the triconnected components algorithm will be useful to us in a form called the Triconnected Components Graph (TCG) that is constructed as follows:

(i) The TCG has a vertex for each triconnected component.

(ii) For each pair of virtual edges created by the same split operation, there is an edge joining the vertices of the TCG corresponding to the triconnected components that contain the virtual edge.

The TCG contains in a concise form all the information needed to reconstruct a biconnected graph from its triconnected components. Figure 3.6 shows a biconnected multigraph, its triconnected components, and the TCG derived from them.

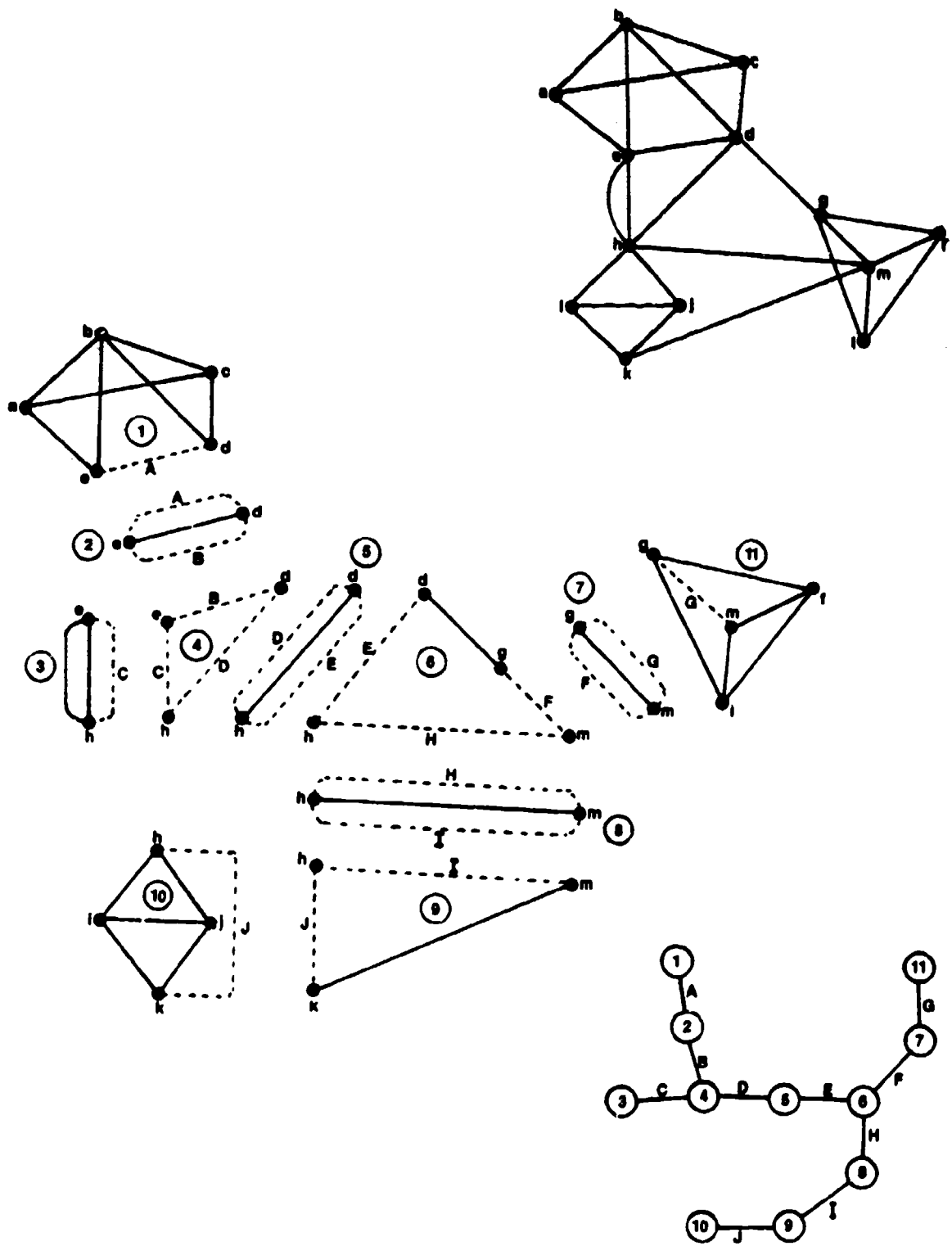


Figure 3.6.

Lemma 3.5.

(a) The TCG of a biconnected multigraph is a tree.

(b) No vertex of a TCG corresponding to a bond (polygon) can be adjacent to another vertex representing a bond (polygon).

Proof. [See Appendix C.] □

3.4 Parsing Two Terminal Networks.

In this section, we explore two applications of the triconnected components algorithm. We will show how the Triconnected Components Graph of a Two Terminal network defines its Trahktenbrot repeated decomposition and also how the same Triconnected Components Graph can be viewed as determining all possible parses of the network using the Universal Replacement System defined in Chapter 2.

We have chosen to present this material in a semi-formal manner because we think that this approach results in a more readable explanation of the principles involved.

In order to understand how the TCG of a TT network describes its Trahktenbrot repeated decomposition, consider the example of Figures 3.7 and 3.8. Figure 3.7 shows a TT network N, its triconnected components and the TCG derived from them, T. Let us consider T as a rooted tree, with the root being A -- the vertex corresponding to the triconnected component of N that includes the return edge.

Consider now the decomposition of N shown in Figure 3.8. The core of the decomposition, N_0, is the root of T, and the components

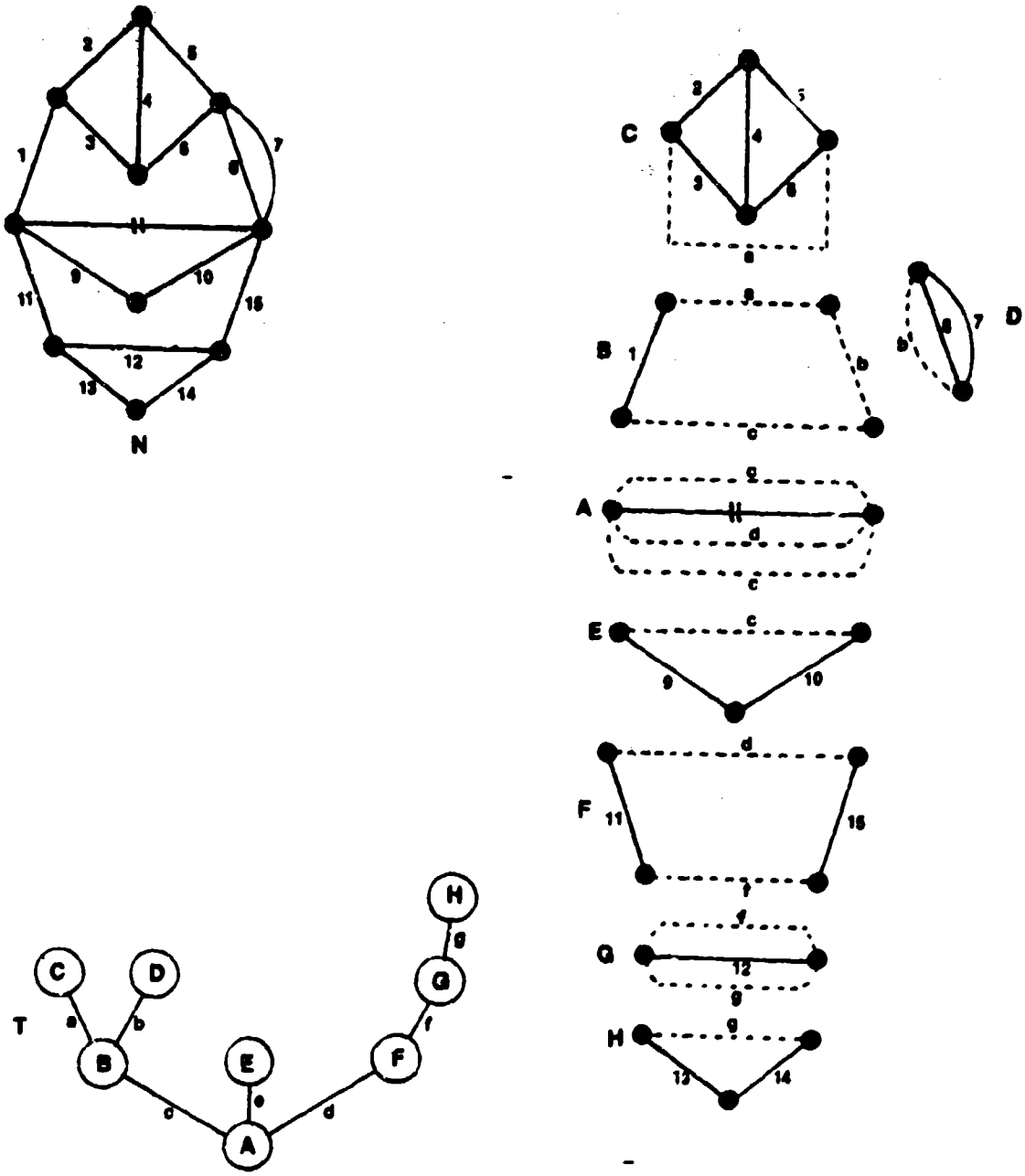


Figure 3.7

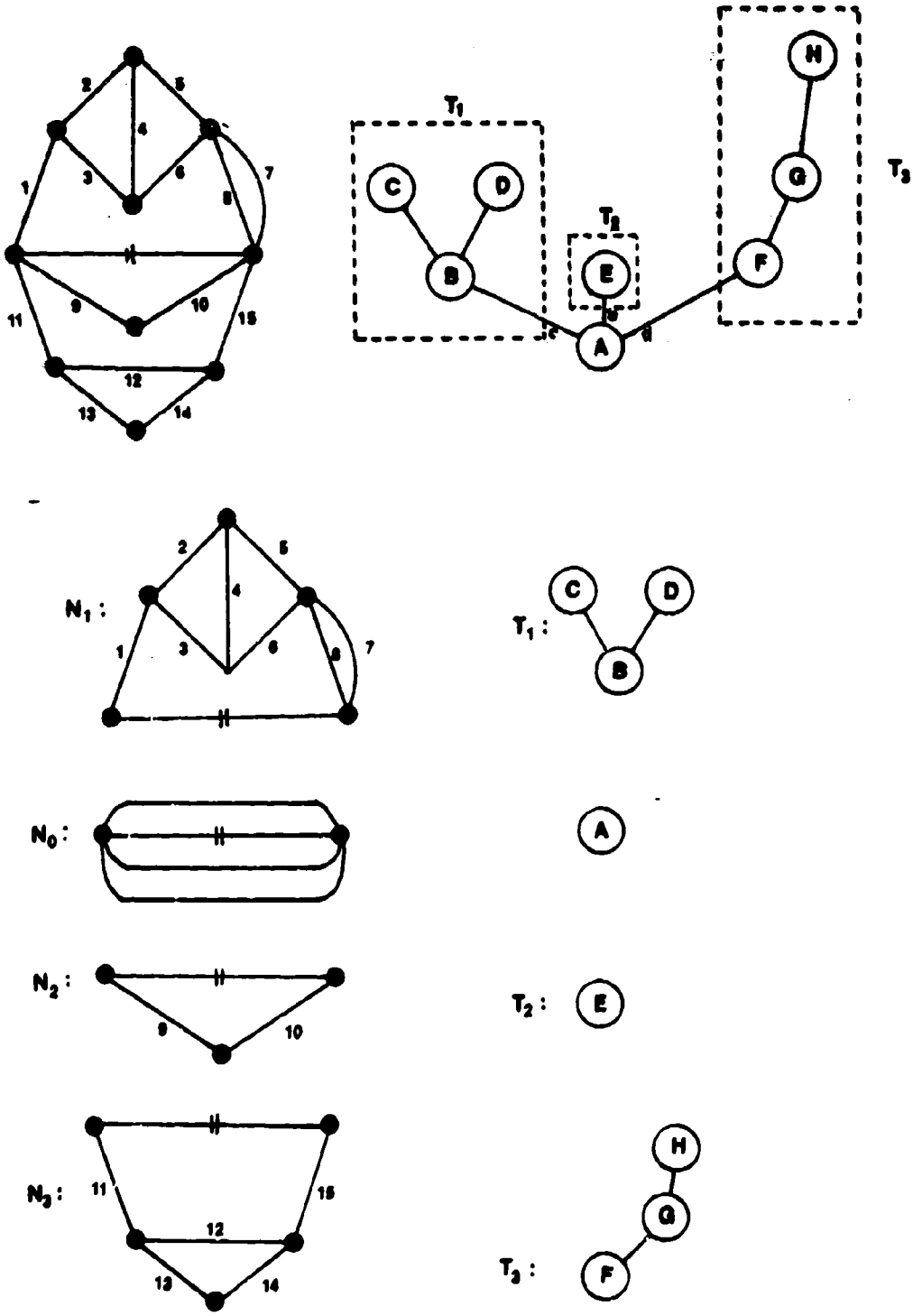


Figure 3.8.

N_1 , N_2 , N_3 , are the graphs obtained by merging the triconnected components of N that fall in the same subtree of the root: T_1 , T_2 , T_3 .

In this way we have found a parallel decomposition of N , therefore N is a p-network. Furthermore, N_1 , N_2 , and N_3 cannot be p-networks or we would have two adjacent bonds in the TCG of N which is not possible according to Lemma 3.5. Thus we conclude that the decomposition of N shown in Figure 3.8 is the unique canonical parallel decomposition of N postulated by Trahktenbrot's theorem.

This argument given for a particular TT network can be generalized in an obvious way to show that the decomposition of any TT network obtained as described above from its TCG is the unique canonical decomposition of the TT network that Trahktenbrot's theorem mentions.

Returning to our example, all one has to do to complete the Trahktenbrot repeated decomposition of N is to apply, in a recursive fashion, the process just described to N_1 , N_2 , and N_3 by using the trees T_1 , T_2 , and T_3 as TCG's. The result of this process on N_1 is shown in Figure 3.9.

By now it should be obvious that given a TT network N and its TCG, T , (i) it is trivial to obtain the Trahktenbrot repeated decomposition of N from T and (ii) that the set of TT networks resulting from the Trahktenbrot decomposition of N is identical to the set of triconnected components of N .

We turn now to the relationship between the TCG of a TT network and the process of reducing the network to a double bond using the Universal Replacement System defined earlier. Figure 3.10 shows an example of such a reduction process. We will only consider reduction sequences (or parses)

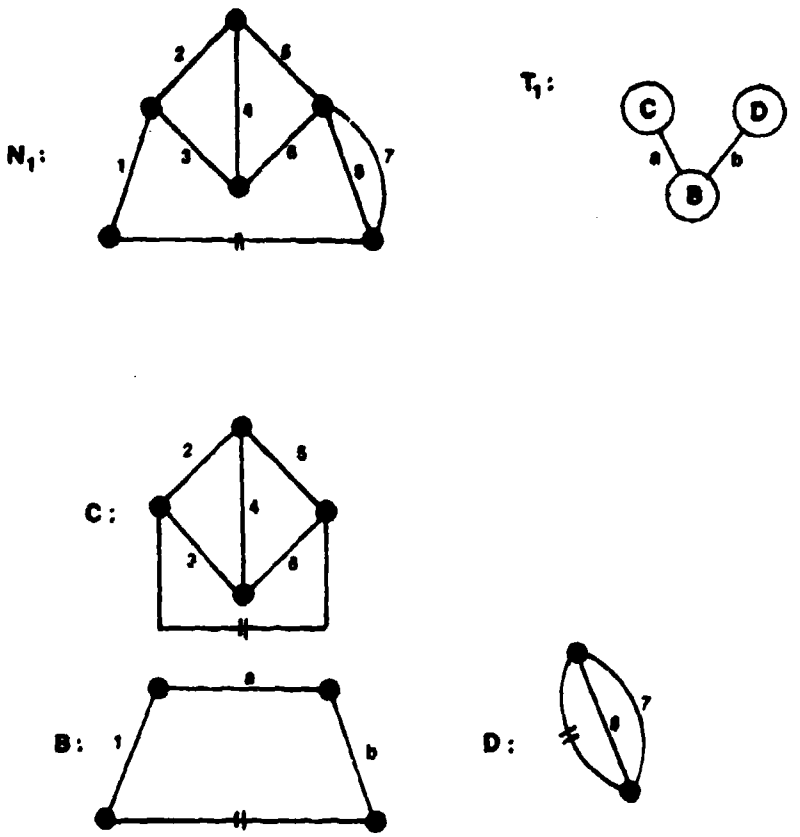


Figure 3.9. The unique canonical series decomposition of N_1 obtained from T_1.

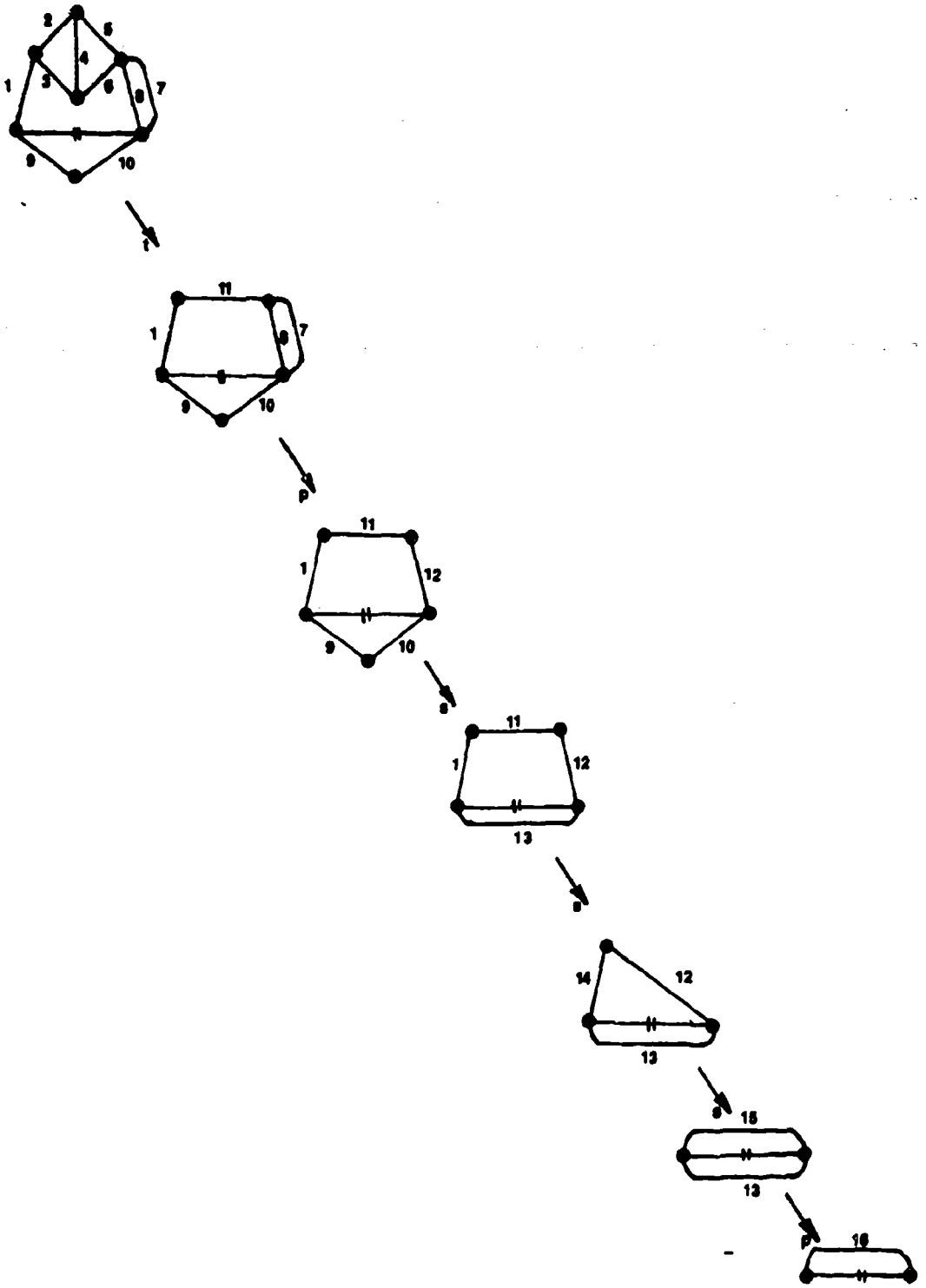


Figure 3.10. Reduction of a TT network to a double bond by series, parallel, and triconnected reductions.

that do not delete the return edge of the TT network. Thus the double bond obtained as an end product consists of the return edge of the original network plus another edge that arises from the reduction.

Any firmly connected TT network can be reduced to a double bond by series, parallel, and triconnected reductions. To prove this, note that each type of triconnected component can be so reduced using only one type of reduction: bonds by parallel reductions, polygons by series reductions, and triconnected graphs with at least four vertices by triconnected reductions. One can therefore perform the process shown in Figure 3.11 in any TT network. This process consists of reducing the TT network by a series of steps, each step replacing the triconnected components of the network that correspond to leaves of its TCG by a single edge. (The TCG is once again considered as a rooted tree.) On each of these steps only one type of reduction is involved.

It is in this way that the TCG of a TT network can be viewed as describing how to parse the network using the Universal Replacement System. There are nevertheless many ways of parsing a TT network and the TCG only describes a few of them, as Figure 3.12 shows. In many applications it is important to have a concise way of representing all the possible parses of a TT network. This goal can be achieved by transforming the TCG into a more detailed structure that we will call a decomposition tree of the TT network. An example of a decomposition tree is shown in Figure 3.13. To construct this tree, each vertex of the TCG has been replaced by a "fan-like" graph as described by Figure 3.14. These graphs have a central vertex labelled "S", "P", or "T" depending on the type of component, a vertex for each actual edge in the component, and a "twig"

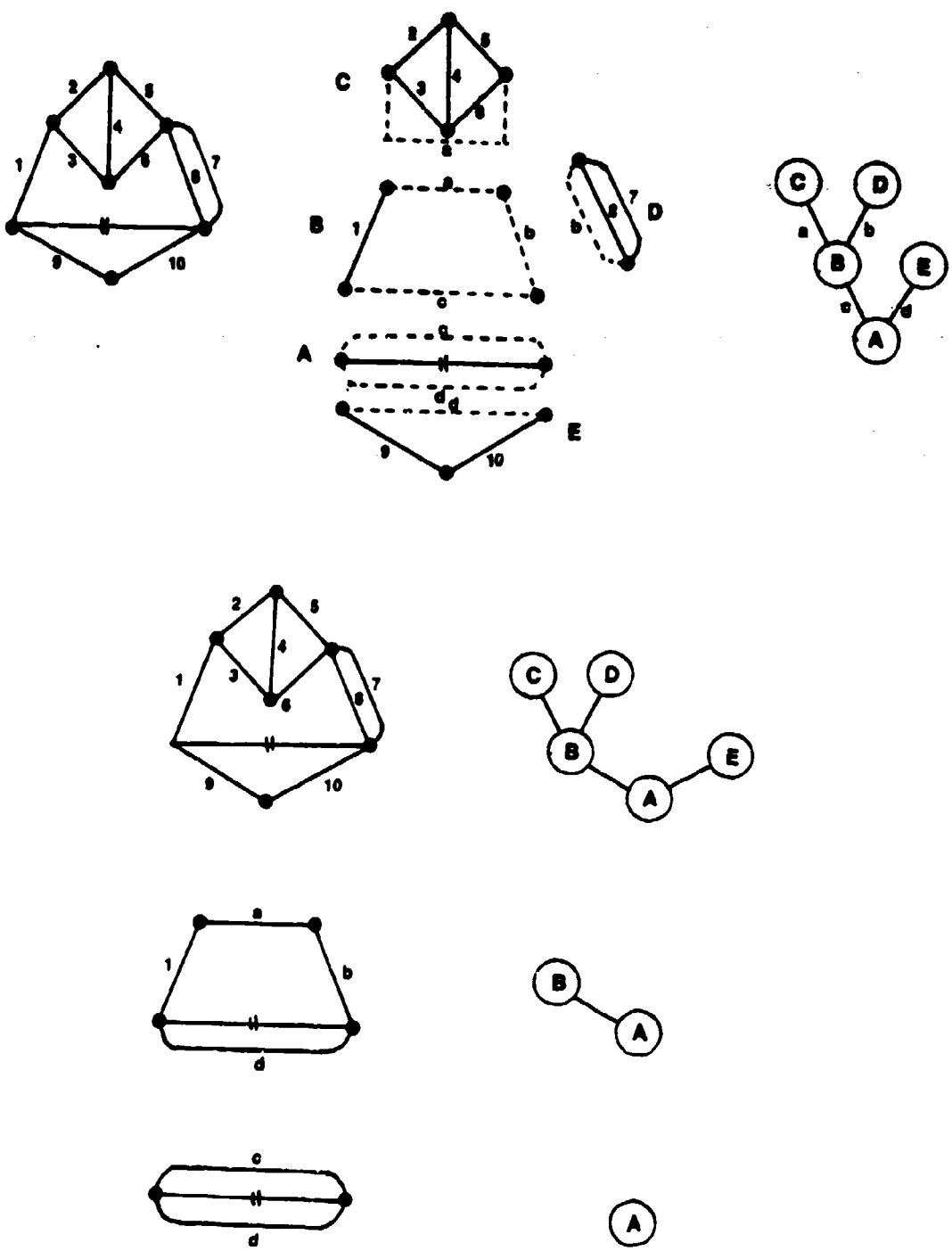


Figure 3.11.

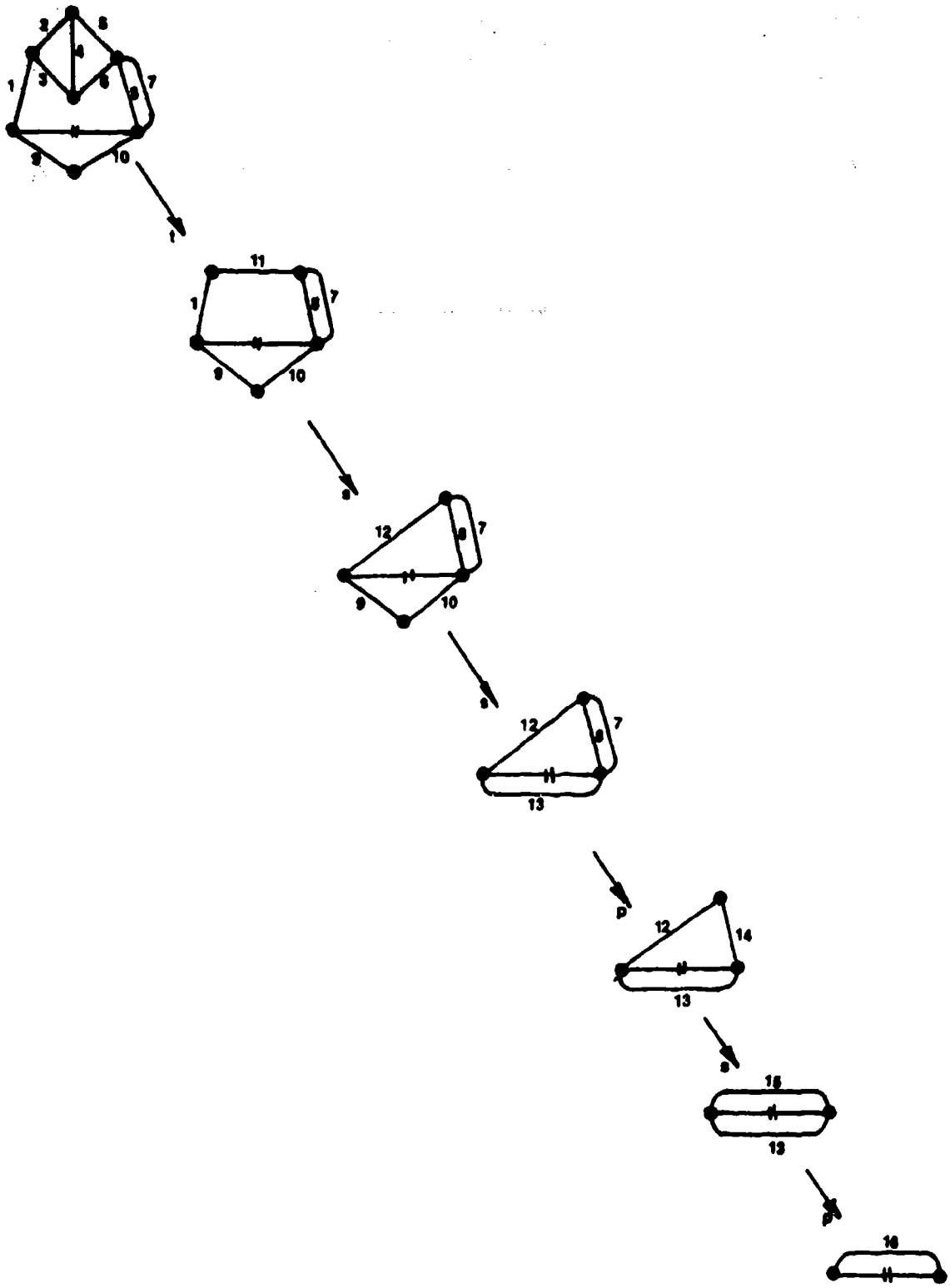


Figure 3.12. A parse of the TT network of Figure 3.11 that is not described by its TCG.

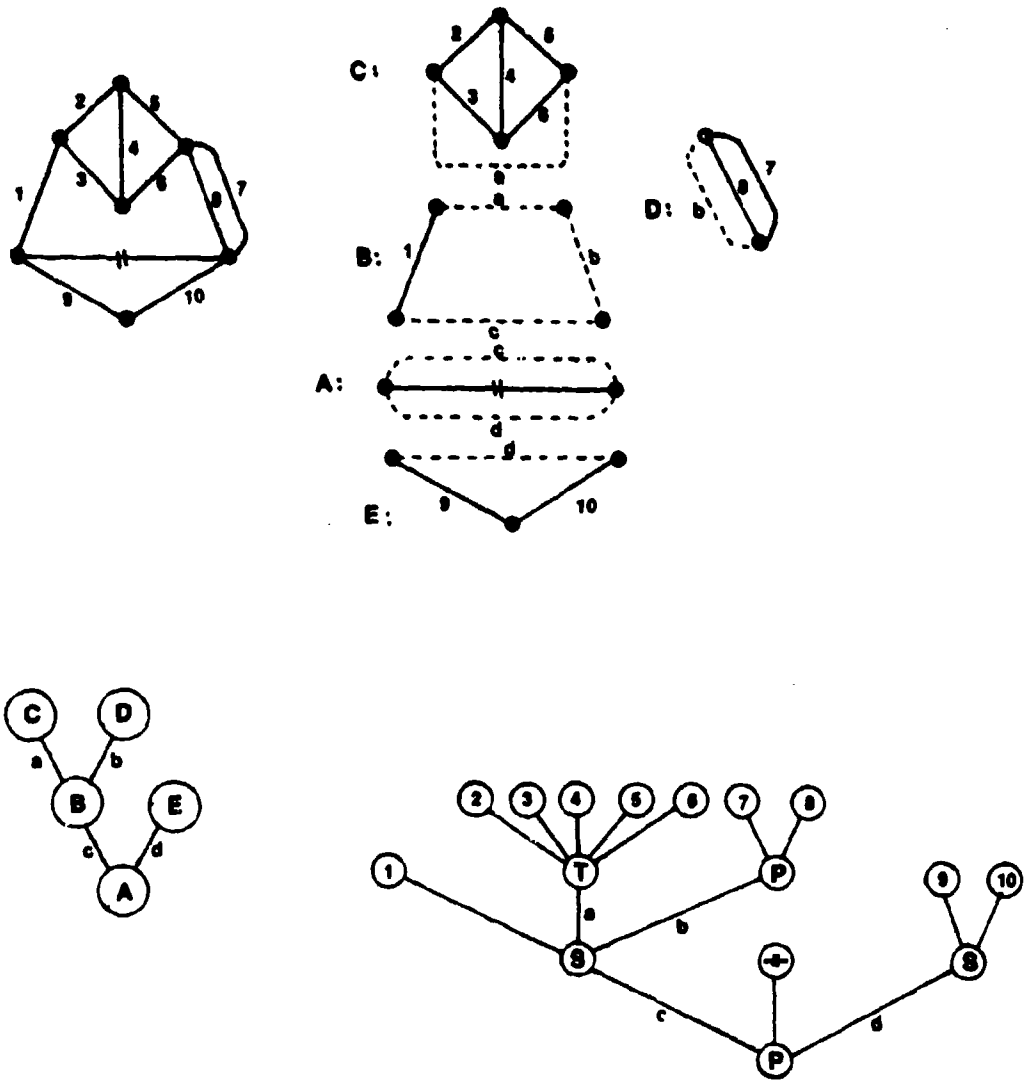


Figure 3.13. A TT network, its triconnected components, its TCG and its decomposition tree.

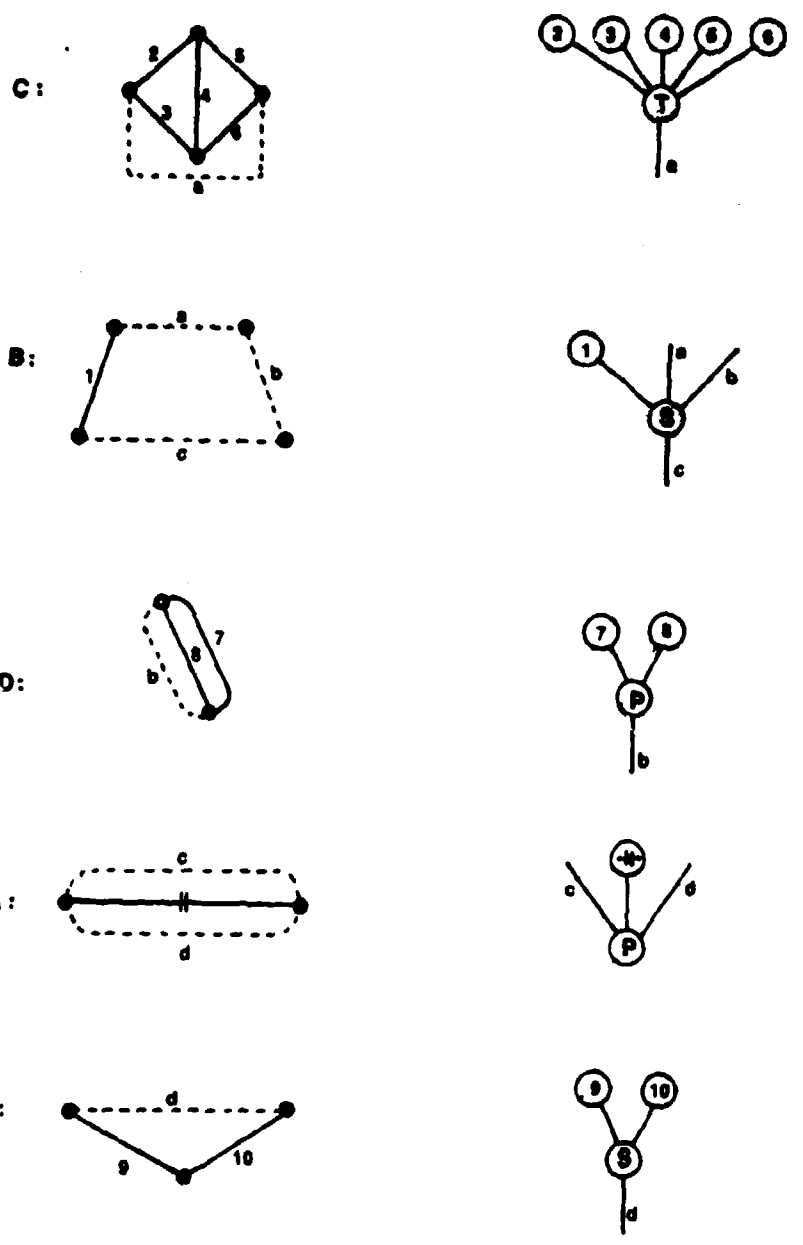


Figure 3.14. The fan-like graphs used to transform the TCG into the decomposition tree in Figure 3.13.

for each virtual edge. The decomposition tree is constructed by "gluing" twigs with the same label.

Using the decomposition tree of a TT network we can describe all possible ways of parsing it by interpreting the reduction rules of the URS as operations on the decomposition tree in the following way:

-- A parallel reduction consists of replacing two leaves that are children of a "P" node by a single leaf. If this operation results in the "P" node having just one child, the "P" node is replaced by its child.
-- A series reduction consists of replacing two leaves that are children of an "S" node and that represent edges of the TT network that have a common endpoint by a single leaf. If this operation results in the "S" node having a single child, the "S" node is replaced by its child.
-- A triconnected reduction consists of replacing a "T" node whose children are all leaves, together with all its children by a single vertex.

As an example, Figure 3.15 shows the parse of Figure 3.12 represented as operations on the decomposition tree of the TT network being parsed. Every parse can be interpreted in this fashion, and therefore the decomposition tree is, as we claimed, a concise representation of all possible parses of a TT network using the URS.

The decomposition tree of a TT network as a rooted unordered tree is unique because it is obtained in an unambiguous way from the set of its triconnected components which is unique. It is important to realize

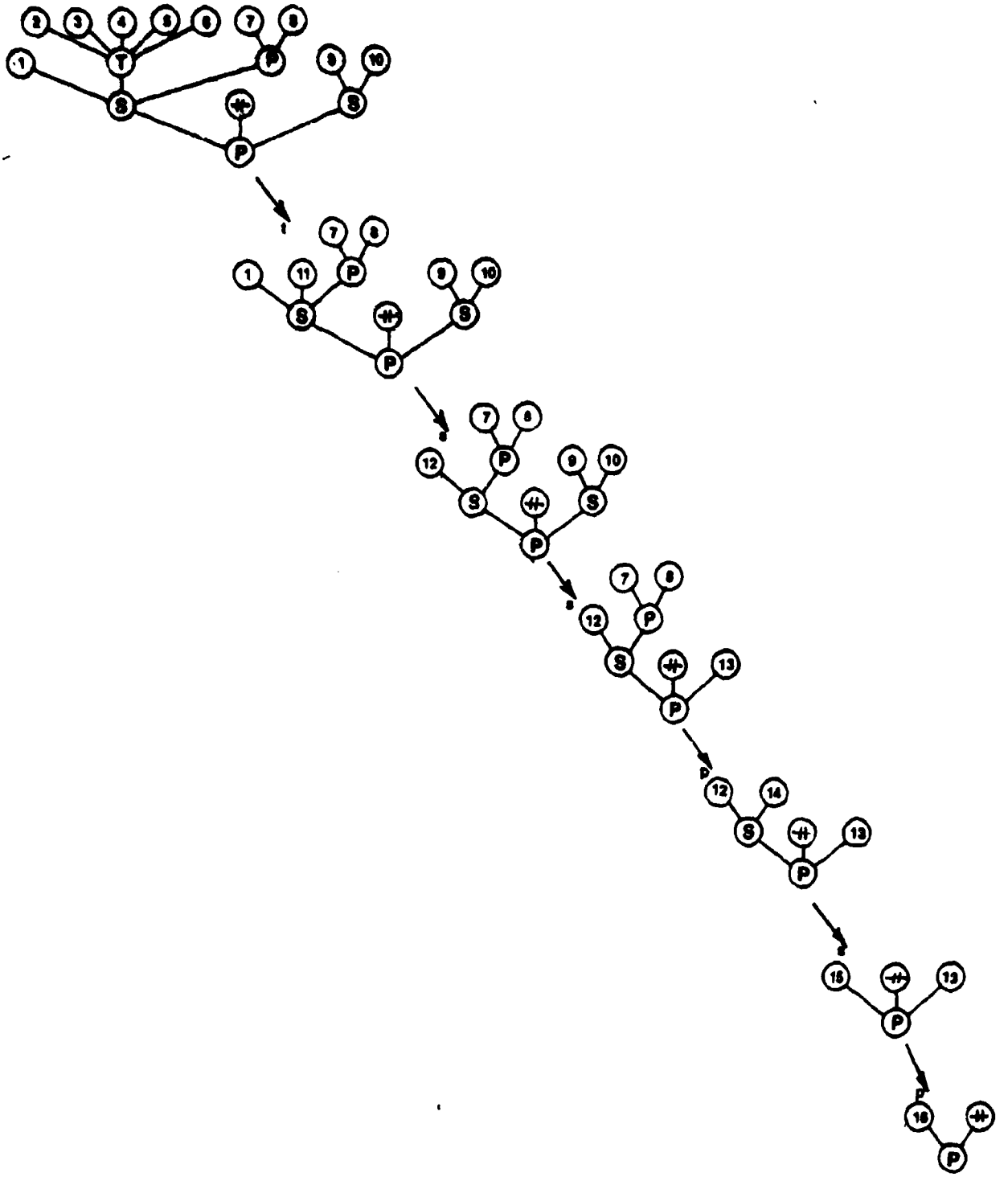


Figure 3.15. The parse of Figure 3.12 represented by operations on the decomposition tree.

however, that, in general, a decomposition tree does not uniquely determine a TT network. In particular, a lot of information is lost in nodes of the tree that represent triconnected graphs with more than three vertices ("T" nodes). The condition underlined in the description of series reduction given earlier (about adjacency of edges) is another example of lost information as one goes from a TT network to its decomposition tree. In the next section we will see that this adjacency information can be translated into a total order of the children of "S" nodes, and as a result, how we can represent in a unique way those graphs whose triconnected component set contains only polygons and bonds using decomposition trees. Among other benefits, this property allows the construction of the decomposition tree of such a TT network from any parse of the network using the SPRS.

3.5 Two Terminal Series Parallel Networks.

We end this chapter on TT networks by studying briefly the class of Two Terminal Series Parallel (TTSP) networks.

We are interested in this class of TT networks for two reasons. The main reason is that a class of directed multigraphs that is very closely related to the class of TTSP networks plays an important role in the algorithm to recognize General Series Parallel digraphs that we will present in Chapter 5. In addition, TTSP networks are the most commonly used and better studied class of TT networks. By unifying some of their most important properties around the concept of triconnected decomposition, we not only provide a theory that is oriented towards the design of efficient algorithms, but also show how most of the classical

results about TTSP networks are related to the general theory of TT network decomposition.

We start this section by defining the class of TTSP networks in an unorthodox manner. We then justify our choice of definition by deriving from it the characterizations used by most authors to define the same class. We end our discussion by exploring the possibility of transforming TTSP networks into multidigraphs (by assigning directions to their edges) in an unambiguous manner, and the most important consequence of this possibility: the one-to-one correspondence between TTSP networks and a modified version of the decomposition trees introduced earlier.

We define TTSP networks in the following way:

Definition 3.1. A firmly connected TT network is Series Parallel if and only if the set of its triconnected components contains only polygons and bonds. □

In a discussion contained in the previous section we showed how any TT network could be reduced to a double bond by series, parallel, and triconnected reductions. In that discussion, each type of reduction was used to transform each of the types of triconnected components (bonds, polygons, and triconnected graphs with at least four vertices) into a single edge. Because of our definition of TTSP networks, it should be intuitively clear that no triconnected reduction needs to be used in the reduction of a TTSP network. The following lemma is thus a direct consequence of Definition 3.1 and the discussion on the reduction of TT networks given in the previous section.

Lemma 3.6. A TT network is Series Parallel if and only if it can be reduced to a double bond by an appropriate sequence of series and parallel reductions.

Proof. [See Appendix C] □

Definition 3.1 immediately suggests a procedure to recognize TTSP networks by inspection of their triconnected components as follows:

Algorithm 3.1 [Recognition of TTSP networks].

Input: A firmly connected TT network: N .

Output: "Yes" if N is TTSP, "No" otherwise.

Step 1: Compute the set, S , of triconnected components of N .

Step 2: If S contains only polygons and/or bonds, answer "Yes", otherwise answer "No". □

This algorithm can be implemented (using Hopcroft and Tarjan's algorithm to implement Step 1) to provide an answer in $O(n+m)$ steps for a TT network with n vertices and m edges.

It turns out that the characterization given by Lemma 3.6 is suitable to be used in an efficient recognition algorithm as follows:

Algorithm 3.2 [Recognition of TTSP networks].

Input: A firly connected TT network N .

Output: "Yes" if N is TTSP, "No" otherwise.

Step 1: Repeatedly apply series and parallel reductions to N that do not delete its return edge.

Step 2: If the multigraph resulting from Step 1 is a double bond, answer "Yes". Otherwise answer "No". □

The correctness of this algorithm follows immediately from Lemma 3.6 and the Church - Rosser property of the Series Parallel Replacement System, but its efficiency depends heavily on how Step 1 is implemented. The problem of implementing Step 1 to run in $O(n+m)$ steps for a multigraph with n vertices and m edges is suggested as an exercise by Aho, Hopcroft, and Ullman in [AHO 72], but unfortunately they do not provide a solution. In the next chapter we will discuss at length the same problem for directed multigraphs. The solution that we will then present can be trivially modified to work on undirected multigraphs, therefore we abandon the problem for the moment.

Our definition of the class of TTSP networks is different from the one used by most authors. The more standard definition, and the one that immediately suggests the applications of TTSP networks, is based on the following characterization:

Lemma 3.7. A firmly connected TT network is Series Parallel if and only if:

(i) it is a double bond, a triple bond, or a triangle,

or (ii) it has a decomposition whose core is a triple bond or a triangle and in which all the components (there are at most two) are TTSP networks.

Proof. [See Appendix C.]

The only way in which the definition given by this characterization differs from the standard definition used by most authors is in the presence of the return edge. We introduced the return edge because it

made the concepts of firm connectivity and biconnectivity equivalent thus simplifying the task of relating the theory of TT network decomposition to the decomposition of a biconnected multigraph into triconnected components.

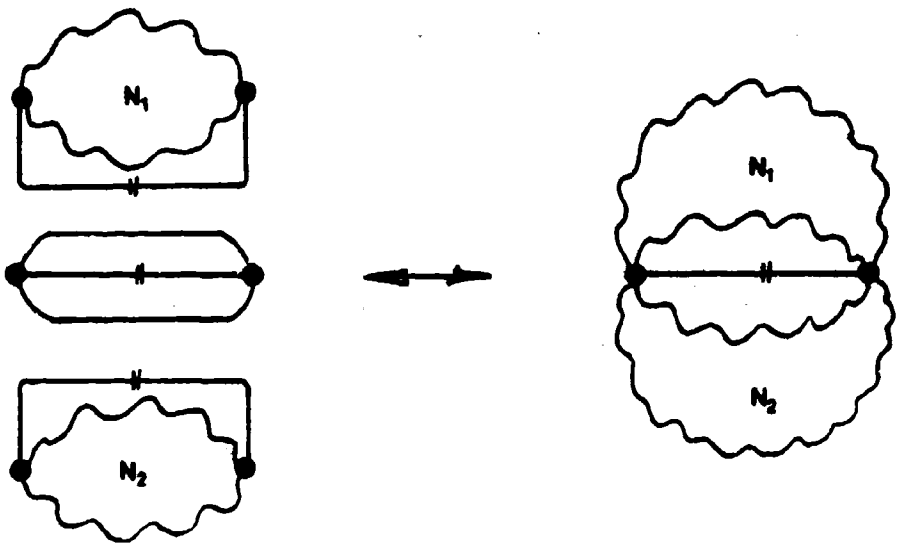
Figure 3.16 shows in a graphic way the relationship between decompositions whose cores are triangles or triple bonds and the familiar operations of connecting two electrical networks in series or in parallel (for more details, see the discussion that follows Definition 4.1 in the next chapter). Lemma 3.7 can thus be interpreted as defining TTSP networks to be the diagrams of all the electrical networks that can be constructed by series and parallel connection of suitable elements like resistors or transistors.

Another common characterization of TTSP networks due to Duffin [DUF 65] that can be easily derived from our definition is the following:

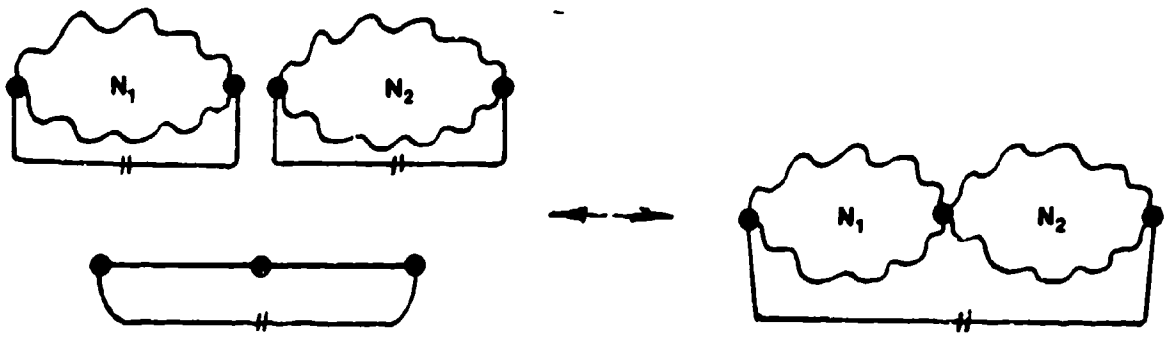
Lemma 3.8. A firmly connected TT network is Series Parallel if and only if it does not contain K_4 (the complete graph on four vertices) as an embedded subgraph.

Proof. [See Appendix C.] □

This characterization is an example of a commonly used method of describing a class of graphs by identifying a subgraph that the members of the class do not contain but every other graph in a wider universe contains. Characterizations of this type are normally called forbidden subgraph characterizations and perhaps the most famous of them is Kuratowskii's characterization of planar graphs (see [HAR 71]). We will provide forbidden subgraph characterizations for some of the classes of



Decomposition with a triple bond as core as parallel connection.



Decomposition with a triangle as core as series connection.

Figure 3.16

digraphs that we will study, and -- whenever possible -- will modify the recognition algorithms for these classes so that when they give a "No" answer they exhibit the forbidden subgraph of their input.

We turn now to two problems that are interrelated: assigning directions to the edges of a TTSP network and devising a unique way of representing any TTSP network by a decomposition tree.

Our procedure to assign directions to the edges of a TTSP network is based on the method of Figure 3.17 to assign directions to the edges of a polygon or bond. That figure shows a polygon and a bond in which exactly one edge has been assigned a direction. We will assign to the remaining edges the directions shown in Figure 3.18: for a bond the directions are such that all edges go in the same direction, and for a polygon they are such that the resulting digraph has a single source and a single sink corresponding to the endpoints of the edge that had a direction originally. Clearly this method can be used to assign directions to the edges of any bond or polygon given a direction for one of their edges.

We now use this method recursively to assign directions to all the edges of a TTSP network using its Triconnected Component Graph as shown in Figure 3.19. We proceed by assigning arbitrarily a direction to the return edge of the TTSP network, then using the method of Figures 3.17 and 3.18 to assign directions to all the edges of the triconnected component that contains the return edge (which is the root of the TCG of the network). In this way we assign directions to the virtual edges that this component shares with several others, and we can use these directions to continue the process going from the root of the TCG towards

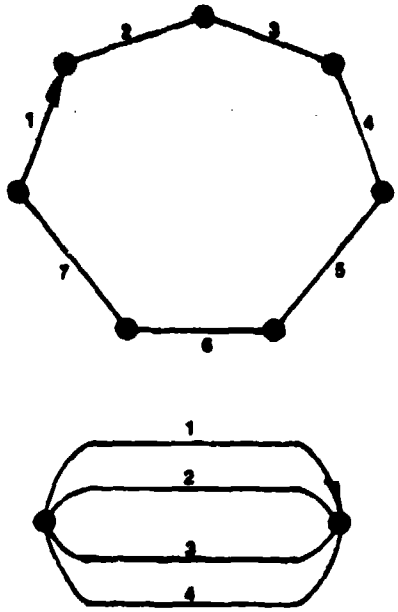


Figure 3.17. A polygon and a bond each having a directed edge.

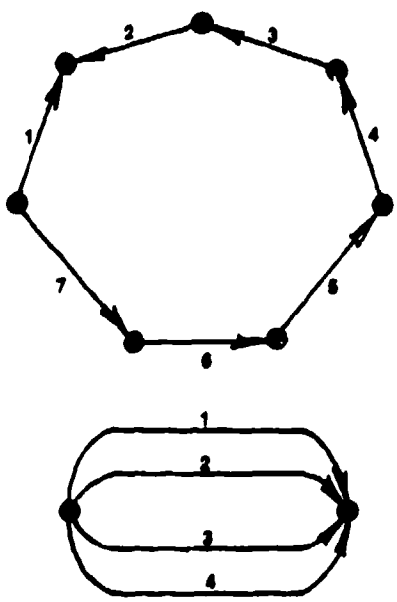


Figure 3.18. Directions implied on the remaining edges of the polygon and bond of Figure 3.17 by their directed edges.

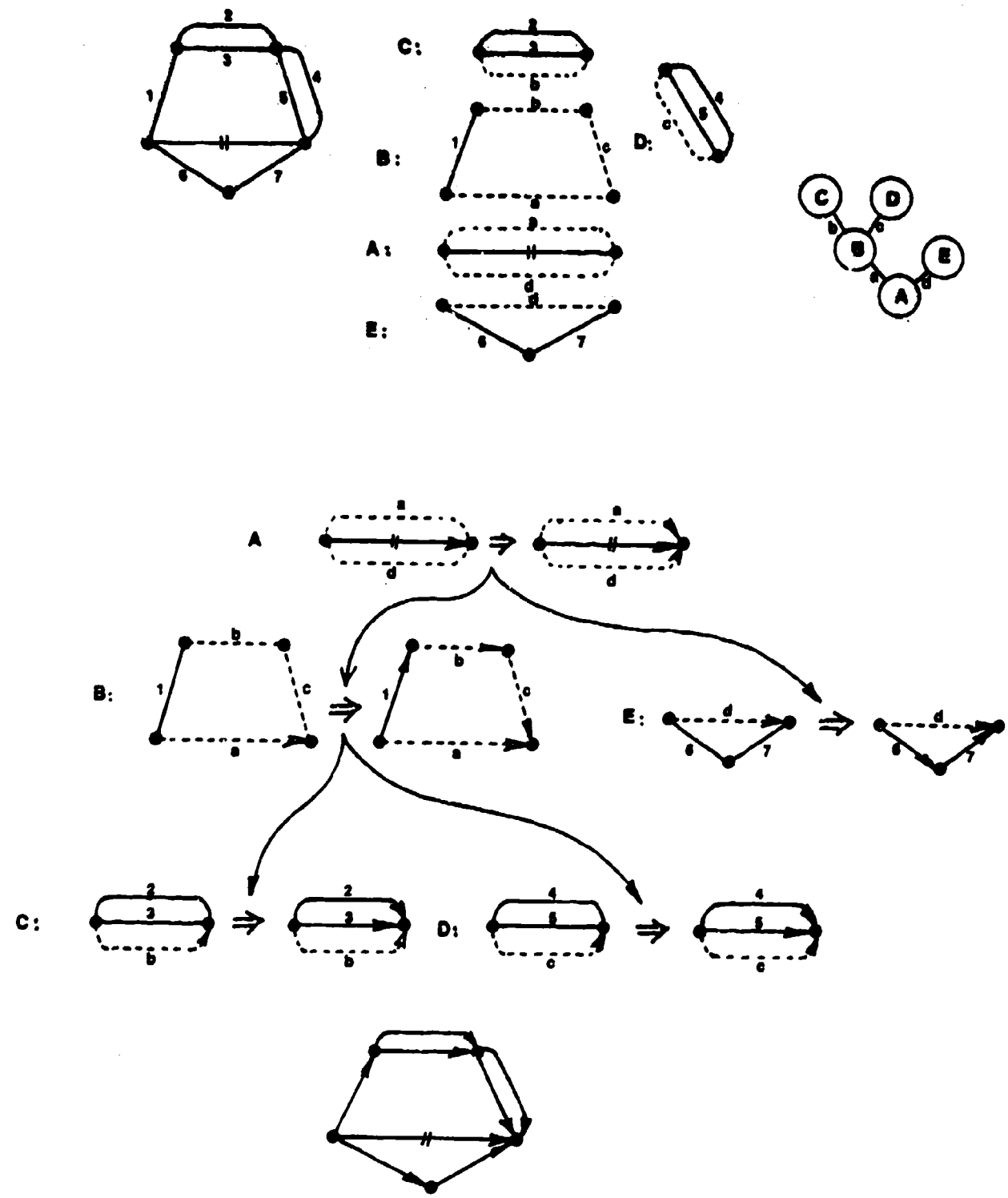


Figure 3.19. Process of assigning directions to the edges of a TTSP network using the TCG and the schema of Figures 3.17 and 3.18.

the leaves. Because the TCG is a tree, this process is unambiguous and the only arbitrary decision that we have made is the direction chosen for the return edge. Since only two directions can be chosen for that edge, and choosing one over the other only reverses all the directions assigned, we have found a quasi-unique way of assigning directions to the edges of a TTSP network. Note that this method relies heavily on the absence of triconnected graphs with more than three vertices from the triconnected component set of the TT network, and is only applicable to TTSP networks. We should also mention that our choice of the return edge to start the process is arbitrary, and any other edge could have been chosen; we chose the return edge to make the discussion similar to the reduction process explained earlier.

Let us now explain how these directions allow the unique representation of a TTSP network by a modified decomposition tree. The key is in the process of translating the triconnected components into "fan-like" graphs as was described in Figure 3.14. For a general TT network these "fan-like" graphs couldn't represent either triconnected graphs with more than three vertices or polygons uniquely. For TTSP networks we only have to worry about polygons, and in this case the problem was that adjacency information was not captured in the fan-like graph as shown in Figure 3.20. The directions of the edges that we described earlier solve this problem by defining a natural total order for all but one of the edges of any polygon, as shown in Figure 3.21. With this total order, a polygon can be uniquely described by an ordered "fan-like" graph as shown also in Figure 3.21. Thus, by considering the children of " S " nodes as an ordered set, we can construct a decomposition tree for any TTSP network that uniquely represents it.

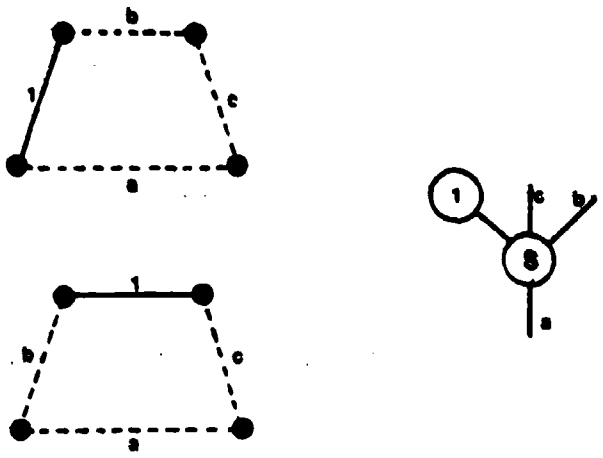


Figure 3.20. Two polygons that are different and can be represented by the same fan-like graph.

Total order induced by directions: l , b , c
Not ordered: a

Figure 3.21. Total order induced by directions on the edges and how to translate it into a unique fan-like graph to represent the polygon.

Figure 3.22 shows this unique decomposition tree for the TTSP network with directed edges of Figure 3.19. It is important to remember that these decomposition trees represent a unique TTSP network only if we consider the children of " S " nodes as an ordered set, and that due to the initial arbitrariness in the process of assigning directions to the edges of a TTSP network, two such trees are possible (one being obtained from the other by reversing the orders of all the set of children of S nodes).

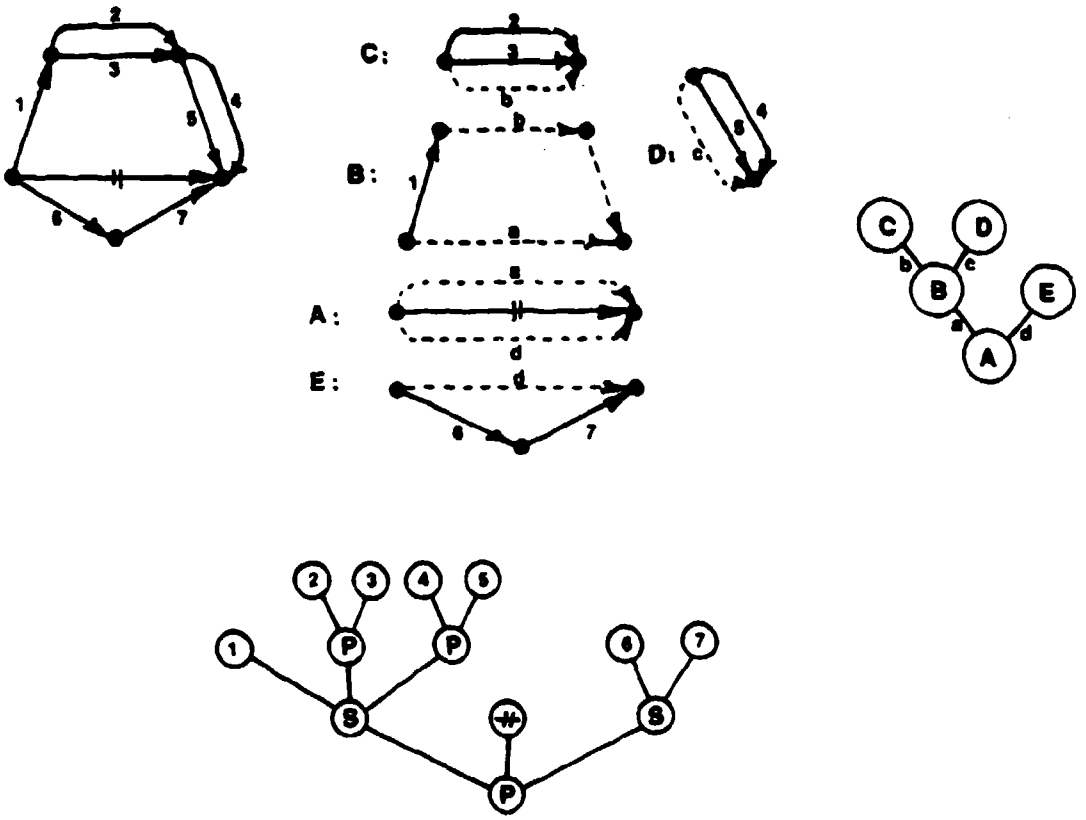


Figure 3.22. Unique decomposition tree obtained from the TCG of a TTSP network and a set of directions assigned to its edges by the process described in Figure 3.19.

Chapter 4. Two Terminal Series Parallel Multidigraphs.

4.1 Introduction.

This chapter can be considered as a natural continuation of the last section of the previous chapter. In effect, the TTSP multidigraphs that occupy us here are nothing more than TTSP networks whose edges have been assigned directions in a specific manner (described earlier) and whose return edges have been removed.

The reason why the material on TTSP networks has not been included in the present chapter is one of emphasis. In the last chapter we were concerned with the relation between the theories of TT network decomposition and triconnected decomposition, and in the last section of that chapter we wanted to show how the basic properties of TTSP networks could be derived from the general theory. In this chapter we study TTSP multidigraphs because we will use them in the recognition algorithm for the class of General Series Parallel digraphs that we will present in the next chapter. Due to this application, the emphasis throughout this chapter will be on the problem of recognizing the class of TTSP multidigraphs and some related problems such as computing the decomposition tree of a TTSP multidigraph or exhibiting the forbidden subgraph of a multidigraph that is not TTSP.

Of the results presented in the last section of the previous chapter only two will be used here. One is the characterization of TTSP netowrks as being reducible by series and parallel reductions. The other is the existence of a decomposition tree that uniquely represents a TTSP network when directions are assigned to its edges in the manner described earlier.

This last fact will be by far the more important, and we will use it indirectly not only in this chapter but in the rest of this work.

The chapter is organized into five sections (not including the introduction). The first is dedicated to a precise definition of the class of TTSP multidigraphs and its relationship to the class of TTSP networks. The next three sections discuss in detail an efficient algorithm for the recognition of TTSP multidigraphs and explain how to modify it so that it (i) returns the decomposition tree of its input whenever it gives a "Yes" answer and it (ii) exhibits a forbidden subgraph on its input whenever it gives a "No" answer. Finally we end the chapter with a section that discusses the use of the unique decomposition trees of TTSP multidigraphs to resolve isomorphism questions.

4.2 Definition and Decomposition Trees.

The class of TTSP multidigraphs (named in this way because all its members have a single source and a single sink) is defined as follows:

Definition 4.1. [Two Terminal Series Parallel multidigraphs].

(a) A digraph consisting of two vertices joined by a single edge is TTSP.

(b) If G_1 and G_2 are TTSP multidigraphs so is the multidigraph obtained by either of the following operations:

Two Terminal Parallel composition: identify the source of G_1 with the source of G_2 and the sink of G_1 with the sink of G_2.

Two Terminal Series composition: identify the source of G_2 with the sink of G_1. □

Figure 4.1 illustrates the operations of Two Terminal Parallel and Two Terminal Series composition and Figure 4.2 shows the construction of a TTSP multidigraph using these operations. The single source and single sink of a TTSP multidigraph are called its terminals.

One brief look at Figures 3.16 and 4.1 should convince the reader that the classes of TTSP networks and TTSP multidigraphs are as closely related as their names seem to indicate. The precise relationship is given by the following lemma.

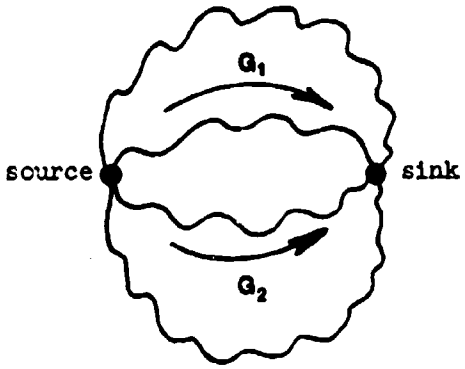
Lemma 4.1.

(a) Let G be a TTSP multidigraph. The TT network obtained by adding a return edge (joining the terminals of G) to the undirected version of G is a TTSP network.

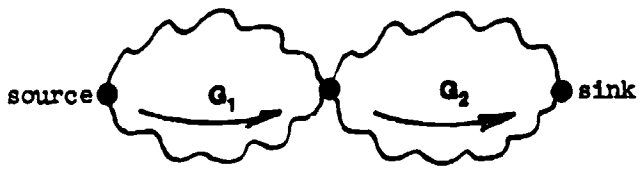
(b) Let N be a TTSP network. The multidigraph obtained by assigning directions to the edges of N as described earlier and deleting the return edge is a TTSP multidigraph.

Proof. [See Appendix C.] □

Given this relationship it is clear that we could have chosen any of the characterizations of TTSP networks given in the previous chapters and use the corresponding version as our definition of TTSP multidigraphs. We chose the recursive definition given because of the resulting ease in proving properties of these multidigraphs using induction. An example of such a property that we will use often (implicitly most of the time) is the following:



Two Terminal Parallel composition



Two Terminal Series composition

Figure 4.1. The operations used in Defintion 4.1.

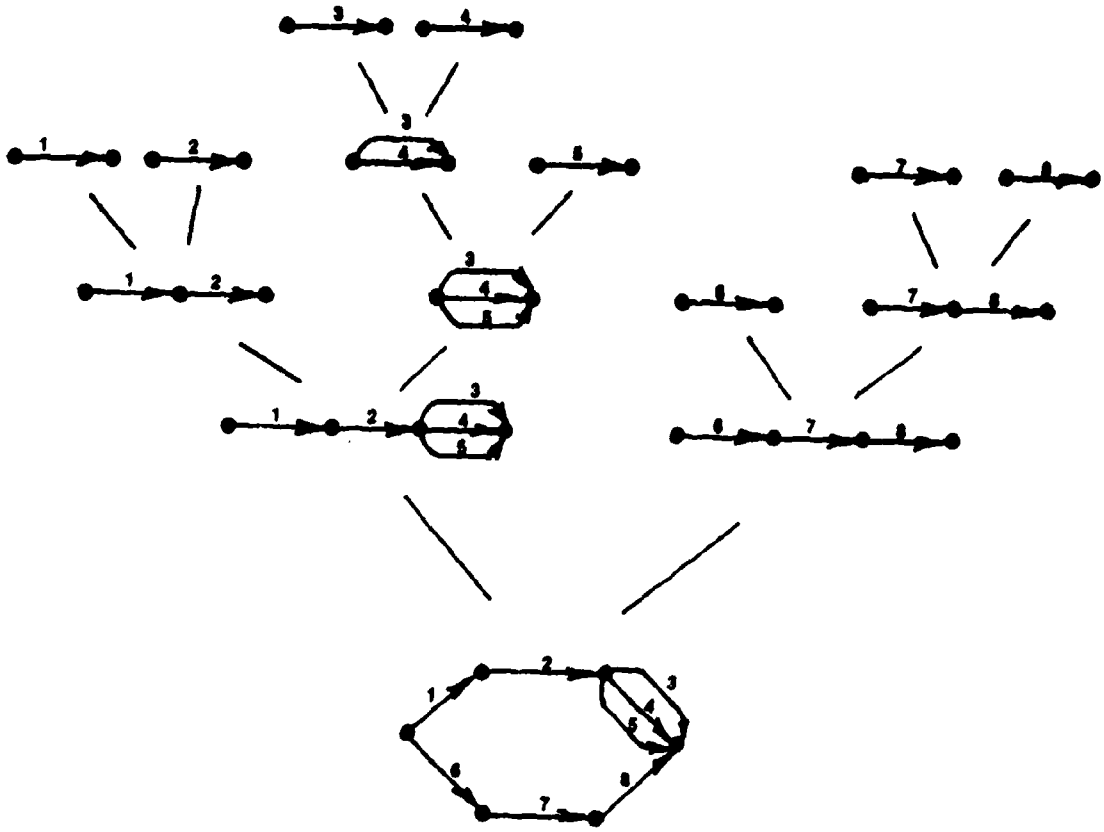


Figure 4.2. Construction of a TTSP multidigraph by Two Terminal Series and Two Terminal Parallel compositions.

Lemma 4.2. TTSP multidigraphs are acyclic.

Proof. [See Appendix C.] □

Let us now consider the problem of representing any TTSP multidigraph by a decomposition tree in a unique way.

The method that we will describe is based on the discussion given in the last section of the previous chapter and is illustrated in Figure 4.3. We start by constructing a TTSP network from the TTSP multidigraphs by ignoring the directions of the edges and adding a return edge joining the terminals. Then we obtain the triconnected components and the TCG of this TTSP network. We then assign directions to all the edges of the triconnected components by assuming that the return edge goes from source to sink of the TTSP multidigraph and using the method described in the previous section. Finally we transform the TCG into a decomposition tree using these directions also by the method described earlier except that no leaf is added for the return edge since it was artificially added to the original TTSP multidigraph.

We will provide no formal proof of the fact that the tree obtained by this process uniquely represents the initial TTSP multidigraph, but it follows from the uniqueness of the triconnected components and the unique way of assigning directions to the edges of these components that we have employed. The only additional fact that one needs to worry about is to show that the directions assigned to the actual edges of the triconnected components coincide with the directions that these edges had in the original TTSP multidigraph. The proof of this fact is implicit in the proof of Lemma 4.1

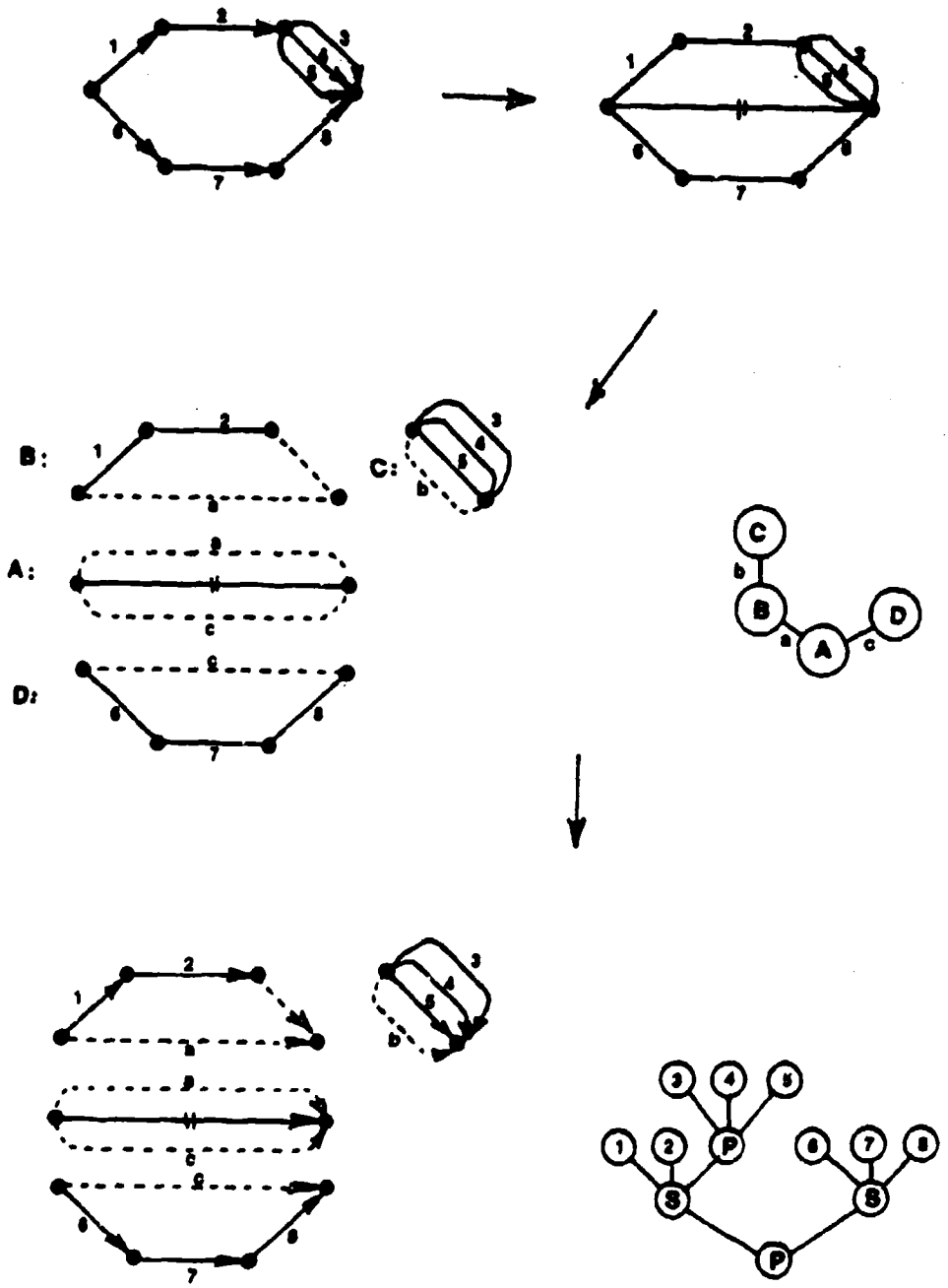


Figure 4.3. How to obtain the unique decomposition tree that represents a TTSP multidigraph.

In the process just described, a certain amount of complication was added by the addition and posterior deletion of the return edge and one may wonder whether this complication was necessary. The return edge plays a very convenient role in the theory of TT network decomposition by making the concepts of firm connectivity and biconnectivity equivalent. On the other hand, when dealing with TTSP multidigraphs the exigence of having an edge joining the terminals is not very reasonable. It is for this reason that we consider the complications that the brief appearance of the return edge on this chapter causes a lesser evil. In any case, the return edge is not used in the next two chapters so we can safely forget about it for the moment.

The construction of a TTSP multidigraphs by Two Terminal Series and Two Terminal Parallel compositions can be very naturally represented by a binary tree as shown in Figure 4.4. In these binary decomposition trees, as we shall call them, the leaves represent edges of the TTSP multidigraph and the internal vertices are labelled " S " or " P " to indicate the Two Terminal Series or Two Terminal Parallel composition of the graphs represented by the subtrees of the node. The order of the children of a node labelled " P " is irrelevant because of the symmetry of the Two Terminal Parallel composition, but the children of " S " nodes are considered ordered with the left subtree representing the multidigraph that corresponds to G_1 in Definition 4.1.

Two non-isomorphic binary decomposition trees may represent the same TTSP multidigraph as Figure 4.5 shows. This multiplicity is due to the associativity of consecutive Two Terminal Series and Two Terminal Parallel compositions. In spite of this multiplicity there is a very simple way of

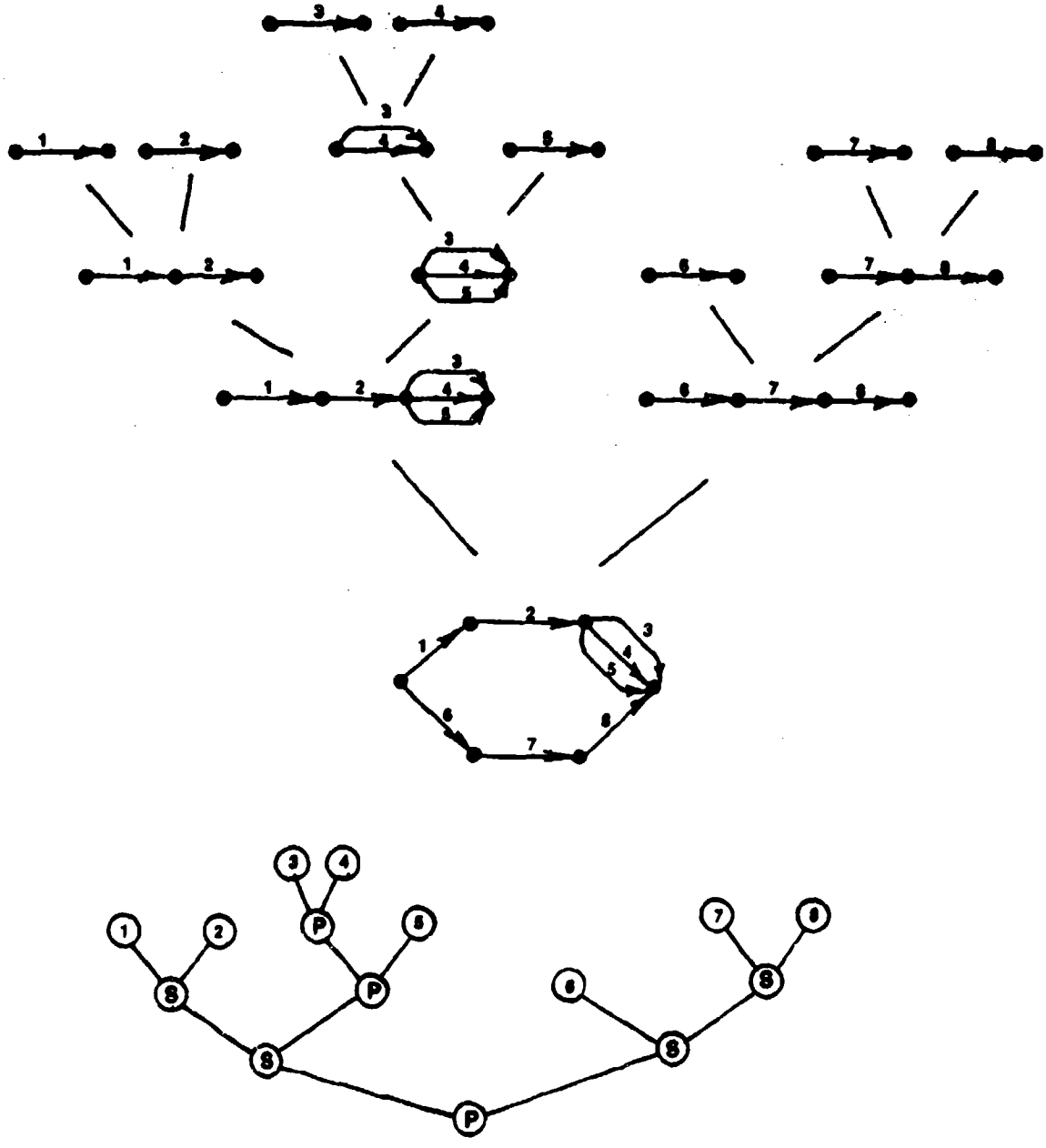


Figure 4.4. Binary decomposition tree for the TTSP multidigraph of Figure 4.2.

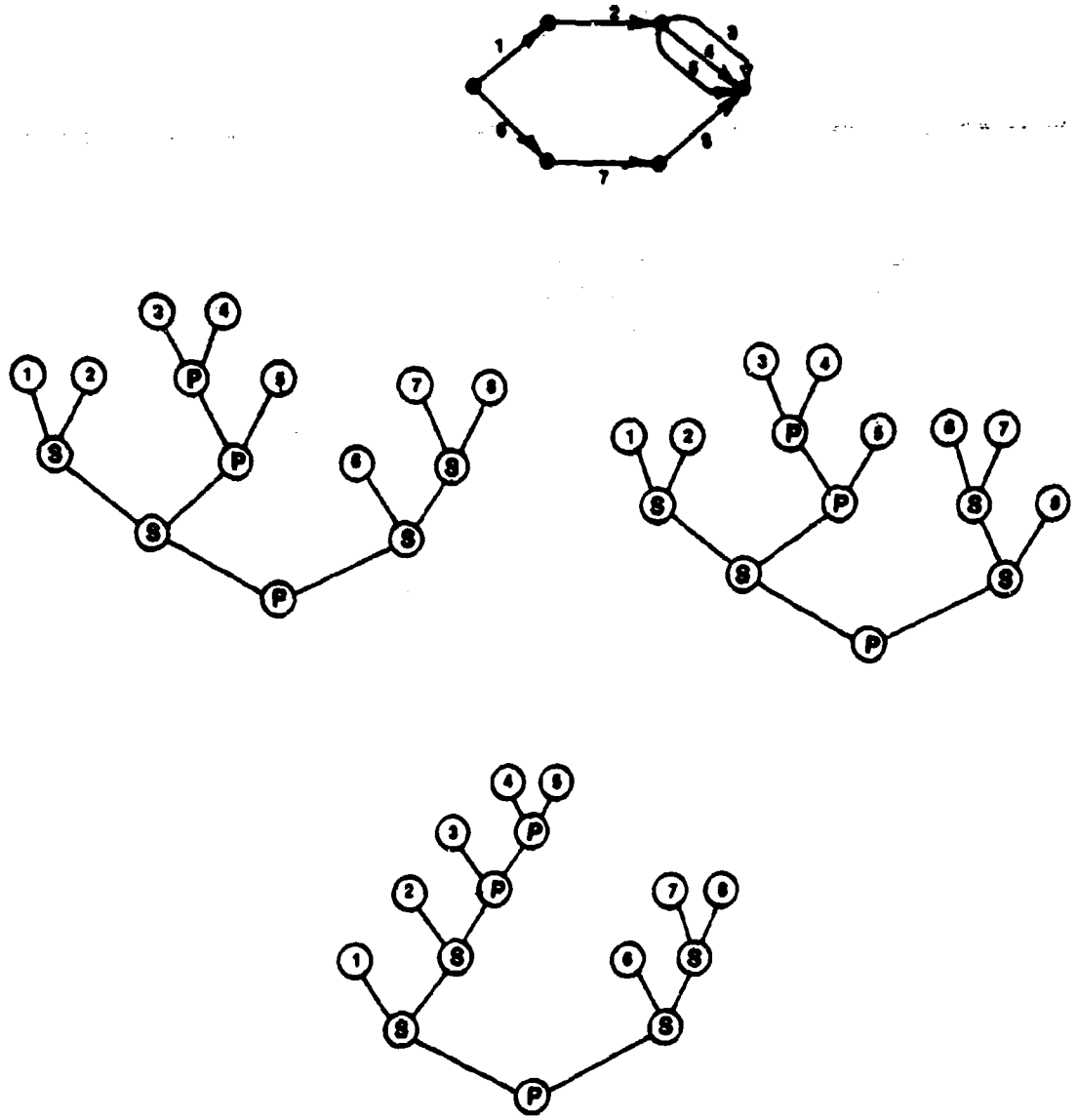


Figure 4.5. A TTSP and several binary decomposition trees that describe its construction.

obtaining the unique decomposition tree of a TTSP multidigraph from any binary decomposition tree. All one has to do is "shrink" the edges of the binary decomposition tree that join internal nodes that have the same label thus identifying their endpoints. This process is illustrated in Figure 4.6.

No formal proof of this relationship is needed if one interprets the unique decomposition trees in much the same way as binary decomposition trees were interpreted: internal "S" ("P") nodes indicate the Two Terminal Series (Parallel) composition of the TTSP multidigraphs represented by the subtrees of the nodes. The only difference is that now a single composition operates on more than a multidigraph (see Figure 4.7). Then it is easy to see how the binary decomposition trees can be obtained from the unique decomposition tree by "associating" the composition operations so each one takes only two arguments.

The above discussion has shown the basic equivalence between binary decomposition trees and the unique decomposition trees (obtained from the triconnected components) for TTSP multidigraphs. For this reason we will use both interchangeably in the rest of this chapter and the following one. Binary decomposition trees are a little more intuitive and easier to manipulate in algorithms, while the uniqueness of the other type of trees will make them very useful to solve isomorphism problems. The reader should keep in mind therefore that when given a decomposition tree of one type with n nodes one can transform it into a decomposition tree of the other type in $O(n)$ steps by a trivial process.

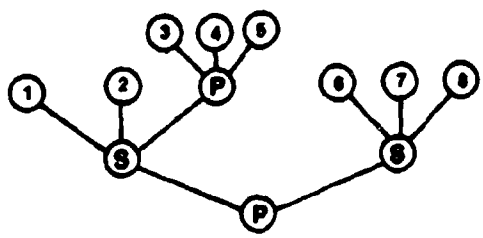


Figure 4.6. The unique decomposition tree obtained from any of the binary trees of Figure 4.5 by "shrinking" the edges that join internal nodes with the same label.

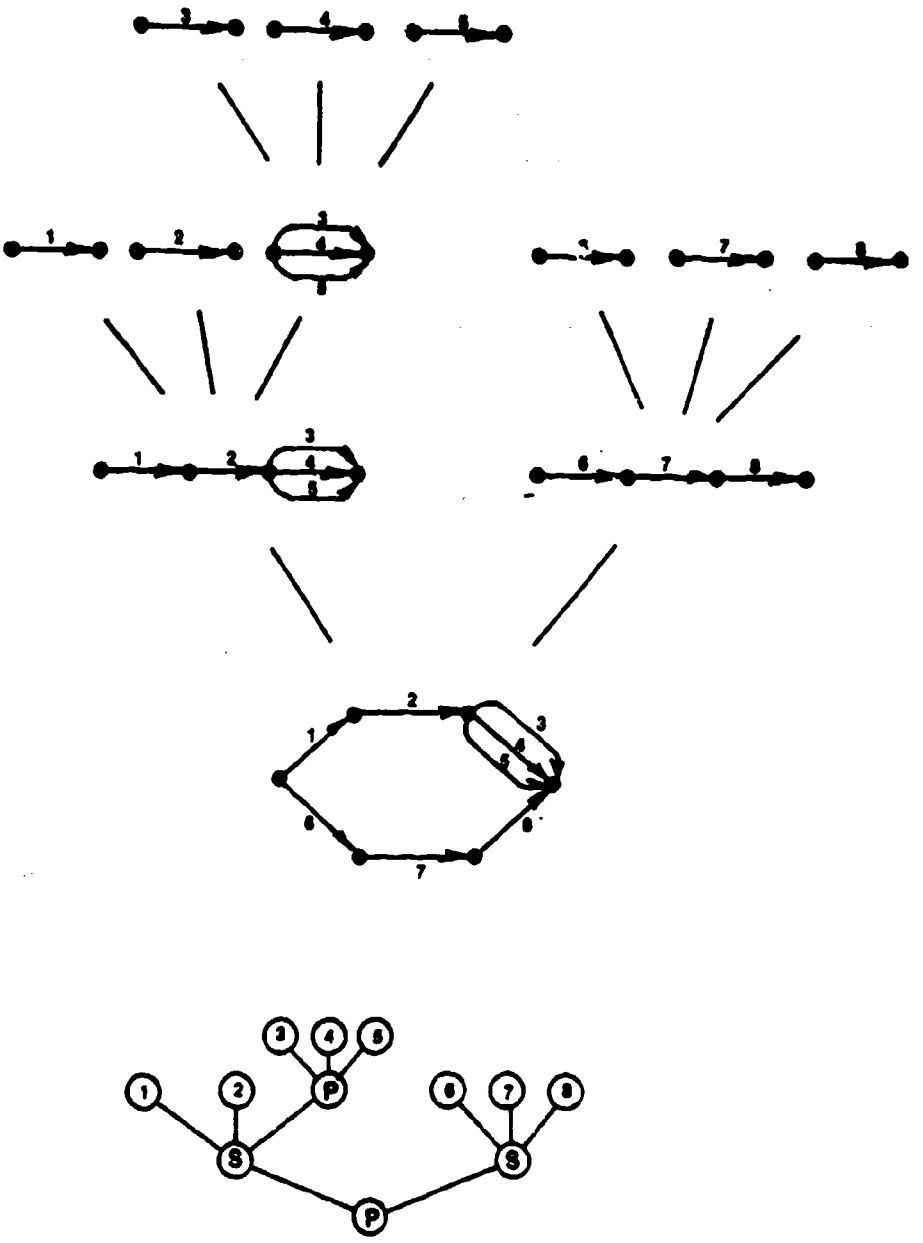


Figure 4.7. The unique decomposition tree of a TTSP multidigraph interpreted as describing the construction of the multidigraph using Two Terminal Series and Two Terminal Parallel compositions that take more than two arguments.

4.3 Recognition of Two Terminal Series Parallel Multidigraphs.

We consider now the problem of recognizing the class of TTSP multidigraphs, that is, given a multidigraph, G, deciding in an efficient manner whether G is TTSP or not.

In the previous chapter we considered briefly the problem of recognizing the class of TTSP networks and outlined two methods to solve the problem. One method was based on the triconnected components algorithm and the other on the Series Parallel Replacement System, and either one can be adapted to work for TTSP multidigraphs. Nevertheless we will only consider in detail the method based on the replacement system because it results in a simpler algorithm.

The characterization of TTSP multidigraphs that we will use corresponds to the characterization of TTSP networks given by Lemma 3.8:

Lemma 4.3. A multidigraph is TTSP if and only if it can be reduced to a single edge by an appropriate sequence of Series and Parallel reductions.

Proof. [See Appendix C.] □

Using this characterization we can test whether a multidigraph is TTSP by the following method:

Algorithm 4.1 [Recognition of Two Terminal Series Parallel multidigraphs]:

Input: Any multidigraph G.

Output: "Yes" if G is TTSP, "No" otherwise.

Step 1: Reduce G by series and parallel reductions until obtaining an irreducible multidigraph G_k.

Step 2: If G_k consists of just two vertices joined by an edge answer "Yes", otherwise answer "No". □

The correctness of this method follows immediately from the Church - Rosser property of the Directed Series Parallel Replacement System (Theorem 2.3) that guarantees that G_k is unique, and from Lemma 4.3 that guarantees that G_k will consist of a single edge if and only if G is TTSP.

We can also place a bound on the number of reductions that we will have to perform: because each reduction decreases by one the number of edges of the multigraph, no more than m reductions can be applied to a multidigraph with m edges. Unfortuantely this is not enough to provide a good bound on the running time of Algorithm 4.1 because it depends heavily on how we search for the applicable reductions. The rest of this section is dedicated to the description of a method of implementing Step 1 to run in $O(n+m)$ steps on a multidigraph with m edges and n vertices.

We represent the input multidigraph by records of two types: each record of one type represents one of the vertices and each record of the second type represents an edge of the multidigraph. The record that represents an edge (u,v) contains pointers to the records that represent u and v and a flag that tells whether the edge is still part of the multidigraph or has been deleted by a reduction. Associated with the record that represents a vertex, v, are two lists of pointers to edges that are incident to v. One of these "incidence lists", called the <u>in-list</u>, contains initially pointers to all the edges that enter v, while the other, called the <u>out-list</u>, contains pointers to all the edges that leave v. In addition the record contains a flag that tells whether the vertex has been removed from the multidigraph by a series reduction or is still part of it.

The basic data structure is a list of vertices that we call the unsatisfied-list. Initially this list contains all the vertices except the source and the sink; in general, a vertex will be on this list only if we have to do some work on it.

We proceed by removing any vertex, u, from the unsatisfied list, "cleaning up" its in-list and out-list (a process described below) and then attempting to remove u from the multidigraph by a series reduction.

This process is repeated until the unsatisfied list becomes empty, at which point we can provide an answer by testing whether all the vertices except the source and the sink have been deleted from the multidigraph and whether all the edges remaining go from source to sink.

The "cleaning up" of an incidence list involves the repeated application of the following rules to the first two elements of the list.

(i) if either element points to an edge that has been removed, delete the element;

(ii) otherwise if both point to edges that have the same endpoints we carry out a parallel reduction involving these edges.

When these rules can no longer be applied, the first two elements of the list point to two edges that have different endpoints and that both enter or both leave the vertex being processed, and we end the process.

A parallel reduction deletes the two edges involved from the multidigraph and adds a new edge to it with the same endpoints and directions as the deleted edges. A series reduction of edges (u,v), (v,w) deletes both edges and adds an edge (u,w) to the

multidigraph. In addition each endpoint of the new edge -- u and w -- is placed on the unsatisfied list unless it is the source or the sink or it is already on the list.

The correctness of this implementation can be proved by the following argument.

If the algorithm gives a "Yes" answer it is because we have managed to transform the input multidigraph into a set of parallel edges joining the source and the sink; this multidigraph can trivially be reduced to a single edge and our answer is correct.

If the algorithm gives a "No" answer we know that (independent of any parallel reductions we may execute) every vertex different from the source and the sink that has not been eliminated cannot be removed unless we first remove some other vertex. Clearly this implies that no additional vertex can be eliminated and our answer is once again correct.

Let us now examine the number of steps that Algorithm 4.1 implemented in the way we just described will take when given a multidigraph with m edges and n vertices as input.

Initially we will have $2m$ pointers to edges because a pointer to any edge (u,v) will appear in the out-list of u and the in-list of v. Series and Parallel reductions add new edges but each decreases the total number of edges of the multidigraph by one because they delete two edges and add one. Thus, no more than $m-1$ new edges will be added since no more than $m-1$ reductions can be carried out before we run out of edges. Therefore we will deal with at most $2m + 2(m-1)$ pointers to edges throughout the running of our algorithm.

The unsatisfied list contains initially $n-2$ elements and vertices are added to the list only when a series reduction is performed. Because series reductions eliminate a vertex, no more than $n-2$ of them could be performed and thus no more than $2(n-2)$ additions will be performed to the unsatisfied list. Therefore, since every time we process a vertex we delete it from the unsatisfied list, we will process at most $3(n-2)$ vertices before the unsatisfied list becomes empty.

The processing of a vertex takes a constant amount of time plus the effort needed to "clean up" its adjacency lists. The "cleaning up" takes a constant number of steps for each pointer deleted plus a constant number of steps to decide that the "clean up" has ended. Since we have $O(m)$ total pointers to edges in all lists, and we only process $O(n)$ vertices we will spend at most $O(n) + O(m)$ steps in the "clean up" part of the algorithm.

We therefore conclude that the algorithm will provide an answer in $O(n+m)$ steps.

4.4 Obtaining the Decomposition Tree of a TTSP Multidigraph.

The recognition algorithm that we have just presented is unsatisfactory in an important aspect. In many cases it is not only important to decide whether a multidigraph G is TTSP or not, but also to compute the decomposition tree of G in the case in which it is TTSP. In this section we will describe a simple modification of the recognition algorithm described in the previous section to output a binary decomposition tree of its input whenever it gives a "Yes" answer.

The method that we will describe could be used to produce either the unique decomposition tree or a binary decomposition tree of any TTSP multidigraph. We have chosen to present how to obtain a binary decomposition tree because it is much simpler to describe, and -- as we saw in the second section of this chapter -- it is a trivial operation to obtain the unique decomposition tree from any binary decomposition tree.

The method works by associating a binary decomposition tree with each of the edges of the multidigraph that is being reduced. Initially every edge is associated with a trivial binary decomposition tree consisting of a single vertex. As new edges are introduced by series or parallel reductions, the binary decomposition trees that we associate with them are computed from the binary decomposition trees associated with the edges being deleted using the rules shown in Figure 4.8. We claim that if a multigraph G is reduced to a single edge, e, the binary decomposition tree associated with e -- if computed according to these rules -- is a binary decomposition tree of the TTSP multidigraph G.

An example of this process is shown in Figure 4.9.

We will not provide a formal proof of the correctness of the procedure just described. Instead we will describe in an informal way the reasons why this method produces the results claimed.

Notice that every edge, e', introduced during the process of reducing a multidigraph G by series and parallel reductions, replaces a certain subgraph G' of G. Because we have managed to transform G' into a single edge e' by series and parallel reductions, G' has to be a TTSP multidigraph. Our method works by associating with each new edge a binary decomposition tree of the subgraph of the input that was

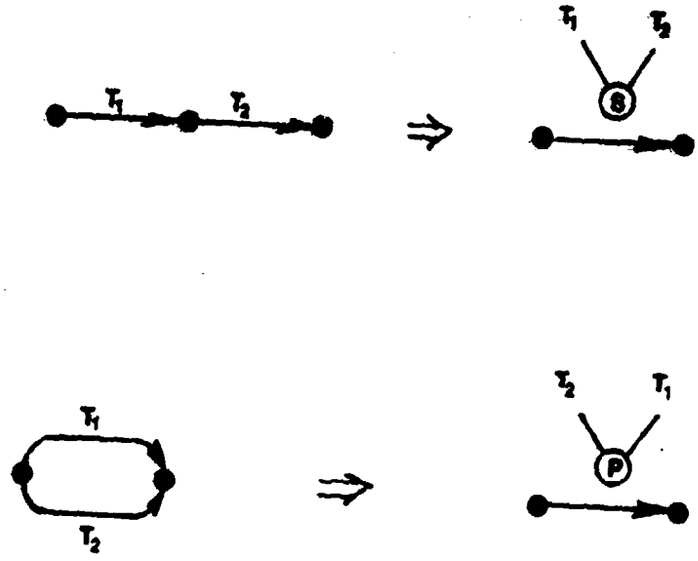


Figure 4.8. Computing the binary tree associated with a new edge from the binary trees associated with the edges it replaces.

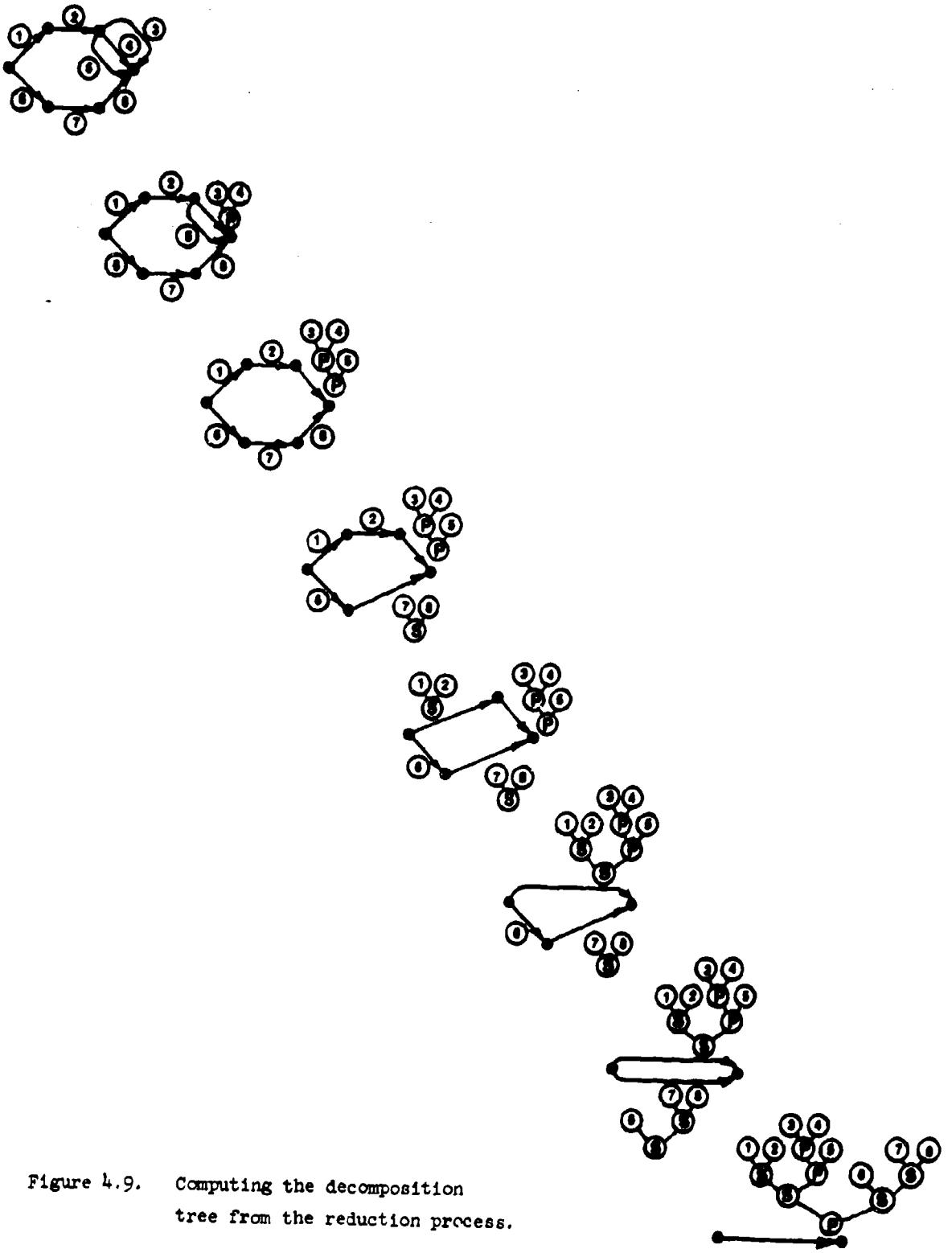


Figure 4.9. Computing the decomposition tree from the reduction process.

replaced by that edge. Following this argument one step further, the binary decomposition tree associated with the only edge remaining after the complete reduction of a TTSP multidigraph G, has to be a binary decomposition tree for G. This argument can be converted into an inductive proof without major difficulties, but in doing so we feel that the simple principle on which the method is based gets lost in the details of the proof.

Let us end this section by discussing the effect that the additional computation needed to implement this method has on the efficiency of the recognition procedure for the class of TTSP multidigraphs described in the previous section.

Clearly the initial association of trivial binary decomposition trees with each edge can be performed in a constant number of steps for each edge. Furthermore any reasonable implementation of the rules of Figure 4.8 would not compute the binary decomposition trees associated with new edges from scratch so to speak, but would rather combine the binary decomposition trees associated with the edges being deleted. In this manner each new binary decomposition tree can be computed in a constant number of steps. Because we compute a new binary decomposition tree for each series or parallel reduction executed, and at most $m-1$ such reductions are performed when we reduce a multidigraph with m edges, no more than $O(m)$ steps are involved in the computation of the binary decomposition tree and the recognition algorithm will give an answer in $O(n+m)$ steps for a multidigraph with n vertices and m edges.

4.5 Exhibiting the Forbidden Subgraph.

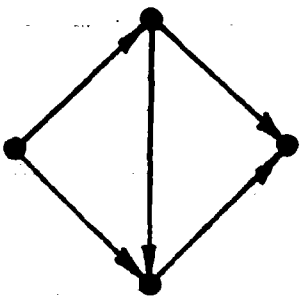
The class of TTSP multidigraphs has a simple forbidden subgraph characterization. This characterization is very similar to Duffin's characterization of TTSP networks (Lemma 3.10) and could be derived from it -- although we will not do so.

Lemma 4.4. An acyclic multidigraph with a single source and a single sink is TTSP if and only if it does not contain the "Wheatstone bridge" (see Figure 4.10) as an embedded subgraph. □

That TTSP multidigraphs do not contain the Wheatstone bridge as an embedded subgraph can be proved easily by induction, showing that the operations of Two Terminal Series and Two Terminal Parallel composition (used to define the class of TTSP multidigraphs) cannot create an embedded Wheatstone bridge by connecting two multidigraphs that do not contain an embedded Wheatstone bridge. In the remainder of this section we will provide an indirect proof of the other half of the above lemma by showing how one can exhibit the forbidden subgraph every time that the recognition procedure gives a "No" answer when given a multidigraph that is acyclic and has a single source and a single sink.

Before proceeding with the proof, let us comment on the requirements of Lemma 4.4: the multidigraph must be acyclic and it must have a single source and a single sink. Figure 4.11 shows why these conditions are not superfluous by displaying two graphs -- each violating one of the conditions -- that are irreducible by series and parallel reductions, do not contain an embedded Wheatstone bridge and are obviously not TTSP.

(a)



(b)

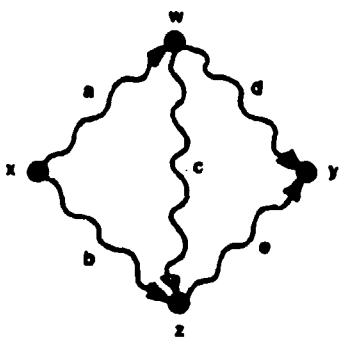
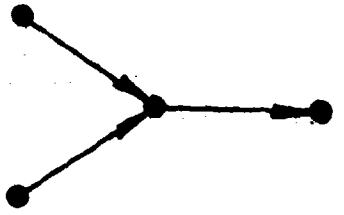


Figure 4.10. (a) The Wheatstone bridge.

(b) A multidigraph contains the Wheatstone bridge as an embedded subgraph if and only if it contains this pattern, where a, b, c, d, and e are disjoint paths.

(a)



(b)

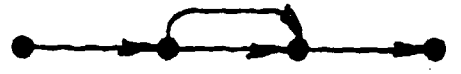


Figure 4.11. Two multidigraphs that are irreducible by series and parallel reductions, are not TTSP and do not contain an embedded Wheatstone bridge.

Let G be a multidigraph that is not TTSP, and let G_E be the multidigraph into which G is transformed by series and parallel reductions by our recognition procedure before it decides that no more vertices of G_E can be deleted and gives a "No" answer. The multidigraph G_E has three important properties: (i) it is an embedded subgraph of G, (ii) it is acyclic if and only if G is acyclic, and (iii) it has the same number of sources and sinks as G. These properties follow directly from the fact that G_E was obtained from G by series and parallel reductions.

Therefore, by counting sources and sinks of G_E and determining whether it contains any cycles we can -- by Lemma 4.4 -- decide whether G_E contains an embedded Wheatstone bridge or not. Because the embedded subgraph is a transitive feature, (that is if G_2 is an embedded subgraph of G_1 and G_1 is an embedded subgraph of G_0, G_2 is an embedded subgraph of G_0) if G_E contains an embedded Wheatstone bridge, so does G. Furthermore, since G_E was obtained from G by series and parallel reductions, the four vertices of an embedded Wheatstone bridge of G_E will be the four vertices of another embedded Wheatstone bridge of G. Therefore the problem of exhibiting an embedded Wheatstone bridge of G can be reduced to the same problem on G_E.

Because G_E was obtained as an endproduct of Algorithm 4.1 running on input G we know that no vertex of G_E can be deleted by a series reduction until some other vertex is deleted first. In other words, each vertex of G_E except its source and its sink has (i) either two distinct successors or (ii) two distinct predecessors or (iii) both. Let us

call a vertex with two distinct predecessors a branch-in vertex and a vertex with two distinct successors a branch-out vertex. The following lemma is the basis of the procedure that we will describe.

Lemma 4.5. There is a branch-in vertex of G_E that is a successor of a branch-out vertex.

Proof. [See Appendix C.] □

Our procedure can be described as follows:

We start by finding the pattern of Figure 4.12 -- which must be present according to Lemma 4.5 -- by examining each vertex of G_E. Once this pattern has been found we use four depth first traversals to find the paths labelled a, b, c, and d in Figure 4.13. These paths have been drawn as if they were disjoint, but in general we wouldn't be so lucky and the paths would have common vertices. We then find vertices u and v such that u is the last vertex of path a which is also on b, and v is the first vertex of c which is also on d.

The new situation is depicted in Figure 4.14 where we know that:

(i) a' and b' are disjoint because of our choice of u.

(ii) c' and d' are disjoint because of our choice of v.

(iii) a' and c', b' and c', and b' and d' are pairwise disjoint or otherwise G_E would contain a cycle.

There are only two cases to consider: that a' and d' are also disjoint or that a' and d' have at least a common vertex. In the first case we have found an embedded Wheatstone bridge and we are done.

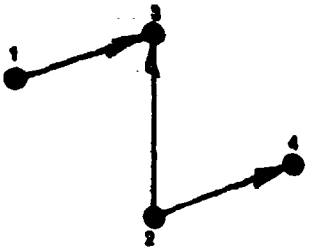


Figure 4.12. A branch-in vertex (3) that is a successor of a branch-out vertex (2).

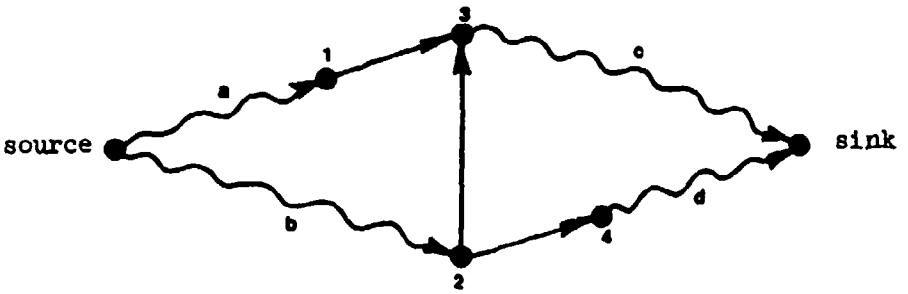


Figure 4.13

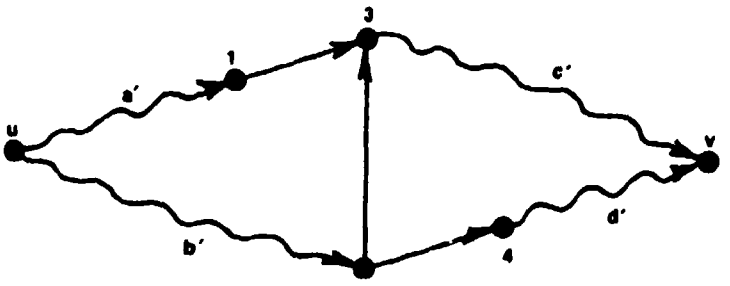


Figure 4.14

In the second case we find a vertex w such that it is the first vertex on a' that is also on d'. If we call a" the section of path a' between u and w, a"' the rest of path a', and d" the section of path d' between vertex 4 and vertex w we have the situation depicted in Figure 4.15 in which all the paths are pairwise disjoint and we have once again identified the embedded Wheatstone bridge.

Let us say a few words about the efficiency of the method just described. We can decide whether G_E contains an embedded Wheatstone bridge by counting sources and sinks and by a depth first search to determine whether it contains any cycle ([TAR 72]). In the rest of the process we need only to find paths between two given vertices, and to find the first (last) vertex of a path that does not (does) belong to another.

Clearly each of these operations can be performed in $O(n_E + m_E)$ steps (where n_E is the number of vertices and m_E the number of edges of G_E) by depth-first traversal. Since we need to perform only a constant number of these operations, we can exhibit the embedded Wheatstone bridge without worsening the asymptotic behaviour of the recognition procedure for the class of TTSP multidigraphs presented earlier.

4.6 Isomorphism of Two Terminal Series Parallel Multidigraphs.

No algorithm is known to resolve the question of whether two graphs with n vertices are isomorphic in a number of steps that grows as a polynomial of n. Nevertheless, efficient algorithms are known for several special cases. Particulary interesting to us is an algorithm that determines whether two rooted trees with n vertices are isomorphic

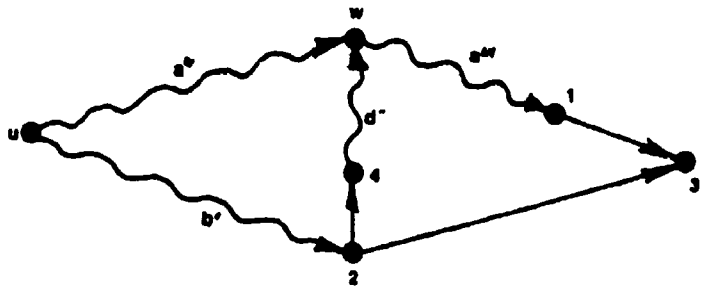


Figure 4.15

in O(n) steps (see Example 3.2 in [AHO 76]), because it can be used to decide the isomorphism of TTSP multidigraphs.

We know that a TTSP multidigraph can be represented by a unique decomposition tree and that we can obtain this tree in O(n+m) steps for a multidigraph with n vertices and m edges. We also know that the decomposition trees are somewhat special in that the children of " S " nodes are an ordered set while the children of " P " nodes are not ordered.

The tree isomorphism algorithm given by Aho, Hopcroft, and Ullman works for unordered rooted trees so it cannot be used immediately to resolve the isomorphism of decomposition trees. This algorithm works by processing the vertices of the trees being tested in levels -- each level contains all the vertices of a tree that are at a fixed distance from the root. The level that contains the vertices farthest from the root is processed first, and then the algorithm works its way towards the root level by level. At each level the algorithm checks that the subtree rooted at each vertex of that level on one of the trees is isomorphic to some subtree rooted at a vertex at the same level on the other tree. The algorithm checks this by assigning a label to each vertex that is computed from the labels of its children and implicitly imposing an order on this set of children by sorting their labels before computing the label of their parent.

The algorithm can be modified so that the label of a vertex whose children are ordered is computed using this order instead of sorting the children by their labels. In this way the algorithm can be used to solve the isomorphism of rooted ordered trees or of trees with mixed nodes like our decomposition trees.

We can therefore determine whether two TTSP multidigraphs with n vertices and m edges are isomorphic in $O(n+m)$ steps using the method just outlined.

A problem much harder than isomorphism is the subgraph isomorphism problem, which consists in determining whether a graph G_1 is isomorphic to some subgraph of another graph G. Clearly, solving this problem implies having solved the isomorphism problem, but having a polynomial algorithm to solve the isomorphism problem does not help much in designing a polynomial algorithm for the subgraph isomorphism problem. The subgraph isomorphism problem is known to be NP-complete (see Exercise 10.9 of [AHO 76]).

Once again an efficient algorithm is known to solve the problem for trees (see [MAT 78] and [DRY 77]), so the question arises of whether we can use this algorithm to determine whether a TTSP multidigraph is isomorphic to a subgraph of another by using their unique decomposition trees.

Unfortunately the matter is not as simple as for the isomorphism problem. Figure 4.16 illustrates the problem. The graph G_1 depicted there is isomorphic to a subgraph of G, but the decomposition tree of G_1 is not isomorphic to any subtree of the decomposition tree of G. Even worse, the decomposition tree of G_1 is not an embedded subtree of the decomposition tree of G. Whether the decomposition trees can be used to design an efficient algorithm for the subgraph isomorphis problem for TTSP multidigraphs remains thus an open question.

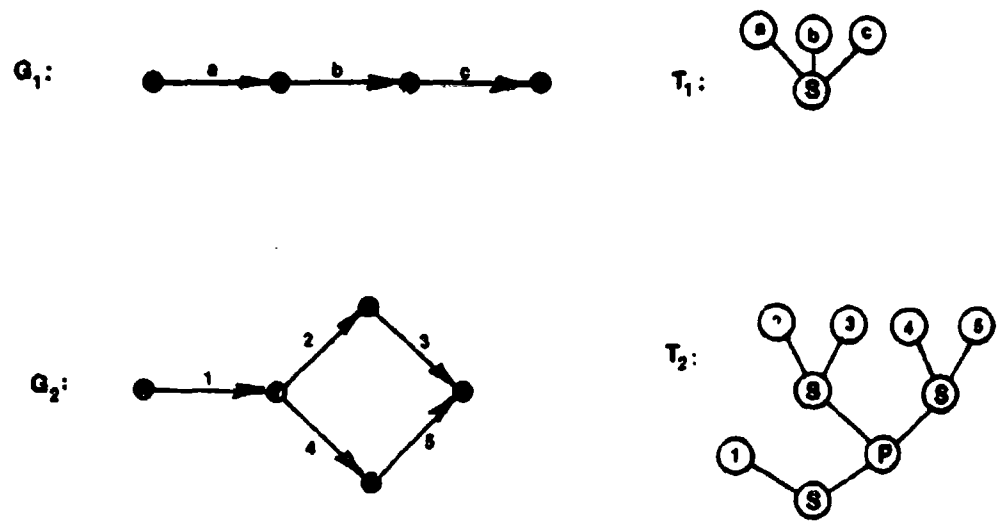


Figure 4.16. Two TTSP multidigraphs and their decomposition trees. G_1 is isomorphic to a subgraph of G_2 but T_1 is not isomorphic to any subgraph of T_2.

Chapter 5. General Series Parallel Digraphs.

5.1 Introduction.

This section is devoted to the study of the class of General Series Parallel (GSP) digraphs.

This class of digraphs was introduced by Lawler to represent sets of constraints between tasks in scheduling problems. For a number of problems of this kind -- some of which are known to be NP-complete for arbitrary constraints -- one can design algorithms that find an optimal schedule for n tasks in $O(n \log n)$ steps when the constraints among the tasks form a GSP digraph by taking advantage of the relatively simple recursive structure of these digraphs. (See [LAW 78], [MOM 77], [SID 76].) All these efficient algorithms use the structure of the constraints to find the solution of a large problem by solving several trivial problems of the same type and then combining the solutions to the trivial problems into a solution for the large problem.

Because many of these optimal scheduling problems have practical applications and because the constraints represented by GSP digraphs arise naturally, it is important to be able to determine efficiently whether a given digraph is GSP, and if it is, to be able to describe its structure in a manner that can be used in the "divide and conquer" strategy used by the efficient algorithms described above. Consequently, the main goal of this chapter is to present an algorithm to perform this recognition task in $O(n+m)$ steps for a digraph with n vertices and m edges.

The remainder of this chapter is organized into five sections. In the first of them (Section 5.2) we provide a formal definition of the class

of GSP digraphs and explore its relationship with the class of TTSP multidigraphs studied in the previous chapter. (This relationship is the basis of another application of GSP digraphs: to the design of hardware specification languages [SMI 78].) The next two sections contain the detailed description of the GSP recognition procedure: in Section 5.3 we describe how to recognize the subset of GSP digraphs that contain no redundant edges, and in Section 5.3 we explain how recognizing this set of minimal digraphs helps us to recognize the class of GSP digraphs.

Section 5.5 introduces a forbidden subgraph characterization for the class of GSP digraphs. The proof of the characterization that we provide consists of a description of how the recognition procedure presented in the previous two sections can be modified so it exhibits the forbidden subgraph whenever it gives a "No" answer.

Finally we end the chapter with a section that considers how the description of the structure of a GSP digraph that our recognition procedure produces when it gives a "Yes" answer, can be used to resolve several questions about GSP digraphs in an efficient manner.

5.2 Definition and Relationship to TTSP Multidigraphs.

We define the class of General Series Parallel (GSP) digraphs in relation to the set of its members that do not contain redundant edges. The members of this set of minimal digraphs are called Minimal Series Parallel (MSP) digraphs and they are defined recursively as follows:

Definition 5.1 [Minimal Series Parallel digraphs].

(i) A digraph consisting of a single vertex and no edges is MSP.

(ii) If $G_1 = \langle V_1, E_1 \rangle$ and $G_2 = \langle V_2, E_2 \rangle$ are two MSP digraphs so is their parallel composition: $G_p = \langle V_1 \cup V_2 , E_1 \cup E_2 \rangle$.

(iii) If $G_1 = \langle V_1, E_1 \rangle$ and $G_2 = \langle V_2, E_2 \rangle$ are two MSP digraphs so is their minimal series composition: $G_{ms} = \langle V_1 \cup V_2 , E_1 \cup E_2 \cup (N_1 \times R_2) \rangle$ where N_1 is the set of sinks of G_1 and R_2 is the set of sources of G_2 . □

The class of GSP digraphs is defined now using the operation of transitive reduction (see Appendix A).

Definition 5.2 [General Series Parallel digraphs]. A digraph is GSP if and only if its transitive reduction is an MSP digraph. □

If we replace the operation of minimal series composition in Definition 5.1 by the operation of series composition, defined by $G_s = \langle V_1 \cup V_2 , E_1 \cup E_2 \cup (V_1 \times V_2) \rangle$, the resulting class of digraphs contains precisely all the GSP digraphs that are transitive. For this reason the members of this class will be called Transitive Series Parallel (TSP) digraphs.

Figure 5.1 shows the construction of an MSP digraph, G_M , by minimal series and parallel compositions. This process can be repeated with the minimal series compositions replaced by series compositions as shown in Figure 5.2 to obtain a TSP digraph, G_T , which is the transitive closure of G_M . The following lemma gives some basic properties of the classes of digraphs just defined and shows that they are related as we claimed:

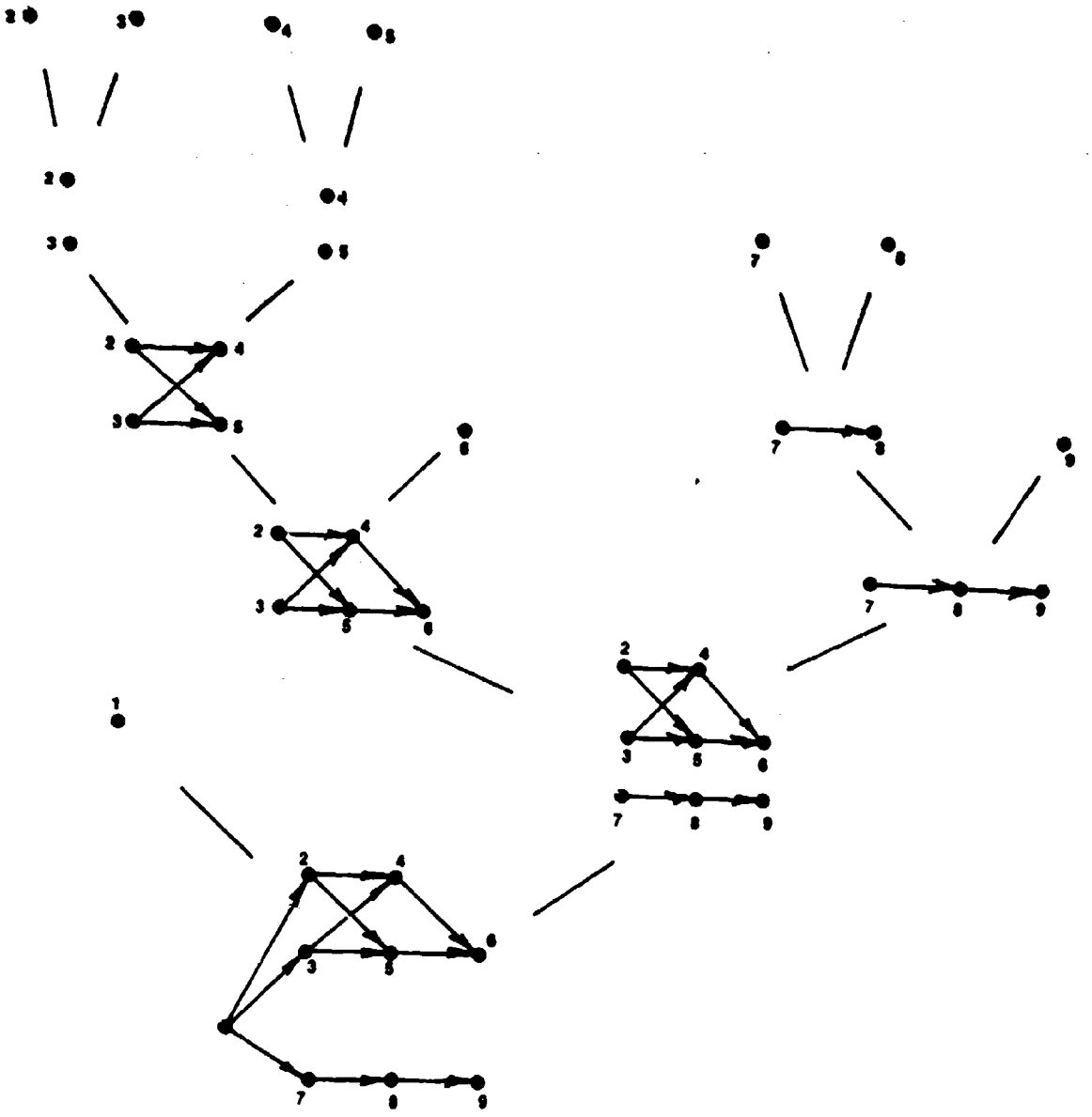


Figure 5.1. Construction of an MSP digraph by minimal series and parallel compositions.

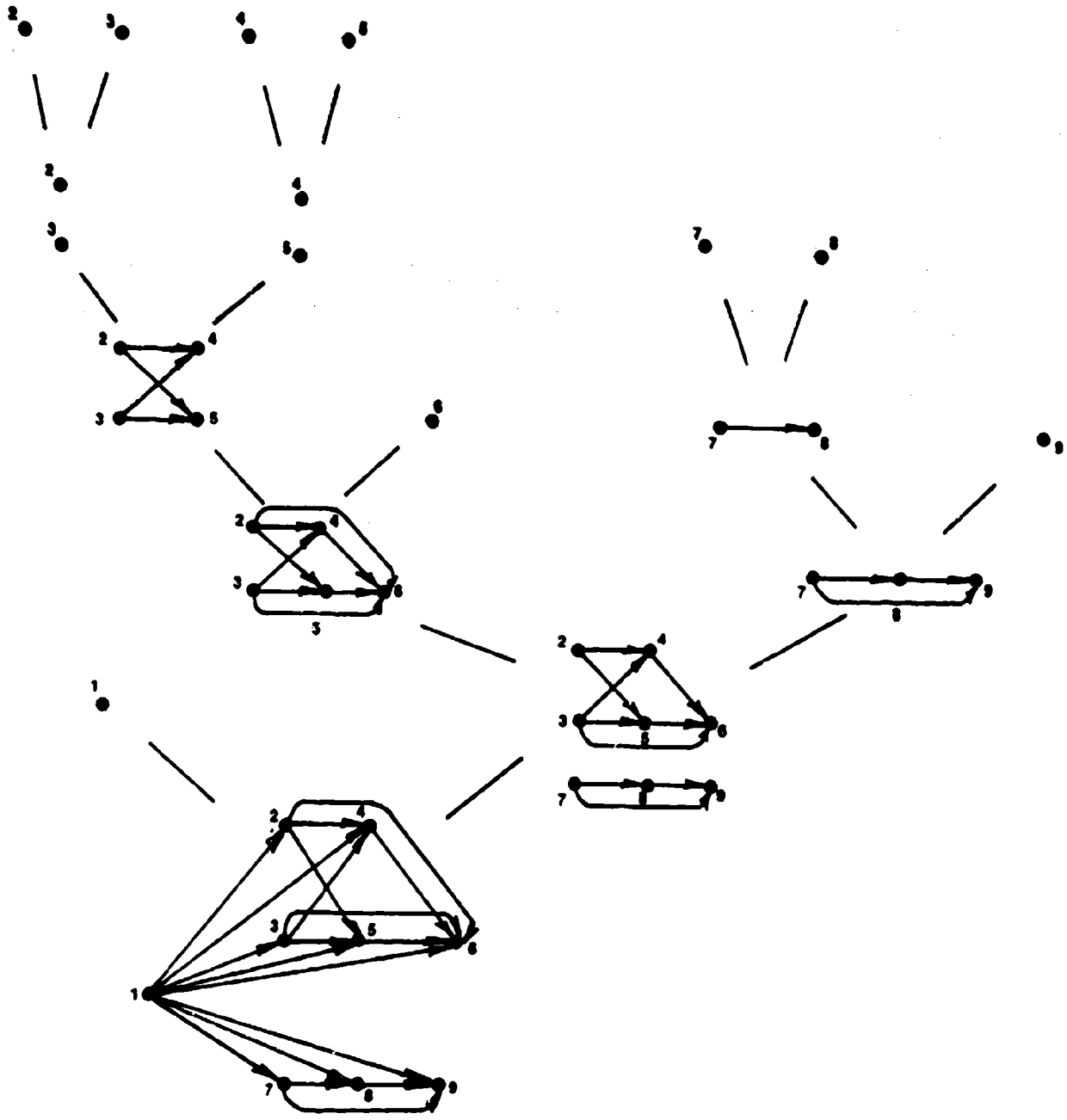


Figure 5.2. Construction of a TSP digraph (the transitive closure of the MSP digraph of Figure 5.1) by series and parallel compositions.

Lemma 5.1.

(i) MSP, GSP, and TSP digraphs are acyclic and contain no multiple edges.

(ii) MSP digraphs are minimal.

(iii) TSP digraphs are transitive.

(iv) The transitive closure of any MSP digraph (and therefore of any GSP digraph as well) is a TSP digraph.

(v) The transitive reduction of any TSP digraph is an MSP digraph.

Proof. [See Appendix C.] □

Note that because of the relationships exhibited by this lemma, the class of TSP digraphs and the operation of transitive closure could have been used to define the class of GSP digraphs instead of the class of MSP digraphs and the transitive reduction operation used in Definition 5.2.

The construction processes of Figures 5.1 and 5.2 can be naturally represented by a binary tree as shown in Figure 5.3. Such a binary tree has a leaf for each of the vertices of the digraph constructed and an internal node for each composition operation used in the construction. The internal nodes are labelled "S" or "P" to indicate respectively the minimal series (series) and parallel composition of the MSP (TSP) digraphs represented by the subtrees rooted at the children of the node. It is important to note that the order of the children of a "P" node is irrelevant (parallel composition is symmetrical) but that the order of the children of "S" nodes is important. We have chosen to represent the digraph that corresponds to G_1 in Definition 5.1 as the left subtree of any "S" node.

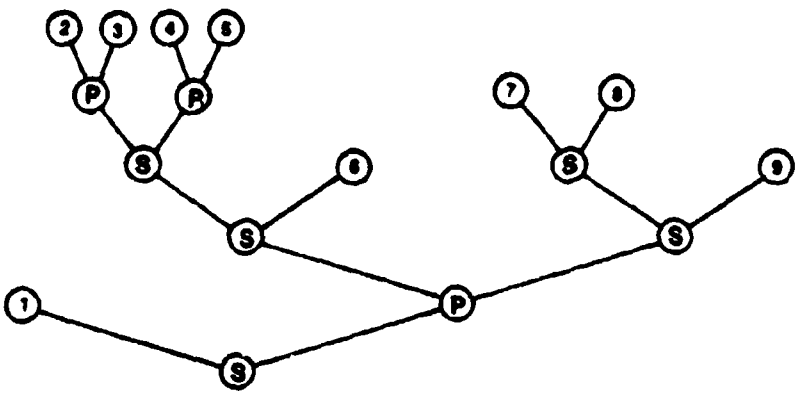


Figure 5.3. A binary tree that represents either one of the construction processes shown in Figures 5.1 and 5.2.

The binary trees just described are formally identical to the binary decomposition trees of TTSP multidigraphs introduced in the previous chapter. For this reason they will be called binary decomposition trees as well.

This formal identity of the binary decomposition trees of TTSP multidigraphs and MSP (or TSP) digraphs is the result of the correspondence between the operations of two terminal series and two terminal parallel composition of multidigraphs on the one hand, and minimal series and parallel composition of digraphs on the other. This correspondence is the following:

Lemma 5.2. Let G_1 and G_2 be two multidigraphs having a single source and a single sink. Let G_{TTS} and G_{TTP} stand respectively for the Two Terminal Series and Two Terminal Parallel compositions of G_1 and G_2, and let $L(G)$ indicate the line digraph of digraph G (see Appendix A for definition).

(i) $L(G_{TTS})$ is the minimal series composition of $L(G_1)$ and $L(G_2)$.

(ii) $L(G_{TTP})$ is the parallel composition of $L(G_1)$ and $L(G_2)$.

Proof. [See Appendix C.] □

Because of this correspondence between the operations used to define the classes of TTSP multidigraphs and MSP digraphs and the identical structure of Definitions 4.1 and 5.1, a one-to-one correspondence can be established between the members of the two classes:

Lemma 5.3. Let G be a multidigraph with one source and one sink. G is TTSP if and only if $L(G)$ is an MSP digraph.

Proof. [See Appendix C.] □

Figure 5.4 shows a TTSP multidigraph and its MSP line digraph as an example of this correspondence.

The correspondence of operations given by Lemma 5.2 implies in a rather direct manner that a binary decomposition tree T can be viewed as representing a TTSP multidigraph or its MSP line digraph. As a result all the properties of the decomposition trees of TTSP multidigraphs proved in the previous chapter can be assumed to be true of binary decomposition trees of MSP (or TSP) digraphs as well. In particular, two non-isomorphic binary decomposition trees can represent the same MSP (or TSP) digraph and we can eliminate this multiplicity by "shrinking" the edges of any binary decomposition tree that join nodes with the same label (see Figure 5.5). In this way, from any binary decomposition tree of an MSP (or TSP) digraph G, one obtains a rooted tree that represents G uniquely. Following the nomenclature of the previous chapter, we will call this unique rooted tree the decomposition tree of G.

We have described how a decomposition tree, T, depending on how it is interpreted, can uniquely represent

(i) a TTSP multidigraph G; or

(ii) an MSP digraph that is the line digraph of G, $L(G)$; or

(iii) the TSP digraph obtained by computing the transitive closure of $L(G)$.

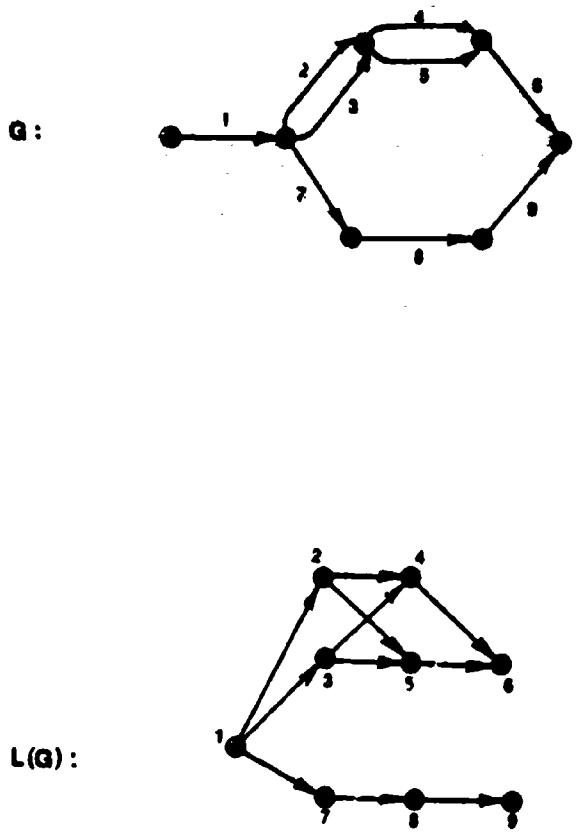


Figure 5.4. A TTSP multidigraph, G , and its MSP line digraphs.

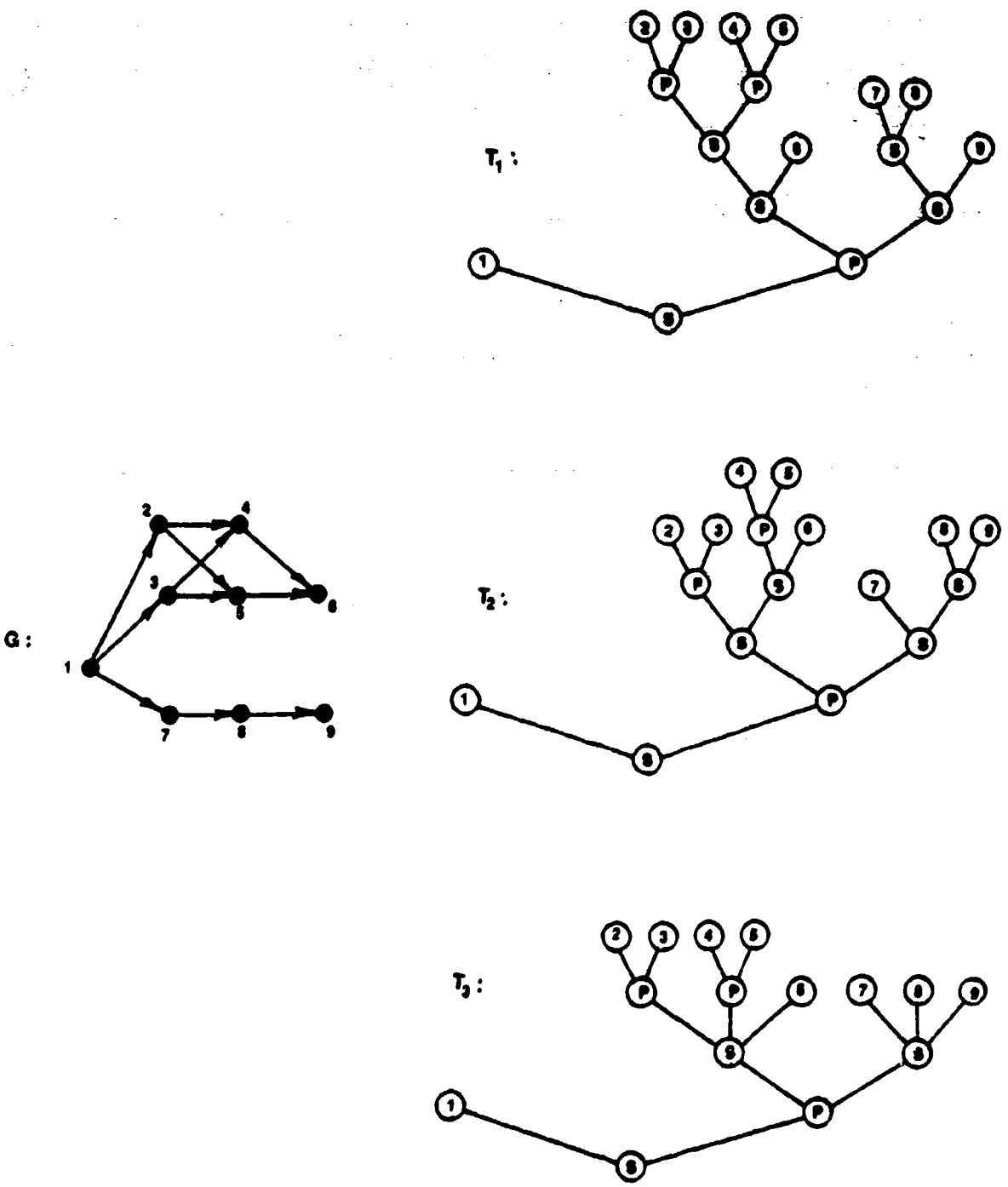


Figure 5.5. Two non-isomorphic binary decomposition trees, T_1 and T_2, that represent the same MSP digraph G, and the unique decomposition tree T obtained from either T_1 or T_2 by "shrinking" edges that join nodes with the same label.

It is important to realize that we cannot represent a GSP digraph by a decomposition tree because there is no natural way to include in the decomposition tree the description of which redundant edges are present and which are absent in the GSP digraph. MSP and TSP digraphs can be represented by decomposition trees because we know that in one case no redundant edge is present and in the other that all the possible redundant edges are present.

The possibility of interpreting a binary decomposition tree in different ways plays a central role in the algorithm to recognize the class of GSP digraphs that we will describe in the next two sections.

This recognition procedure will work as a three step process:

-- On input G, the first step will compute a minimal subgraph G_M of G such that if G was GSP, G_M is its transitive reduction (and therefore MSP).

-- The second step determines whether G_M is an MSP digraph. If it is, the algorithm will compute a decomposition tree, T, of G_M, and if G_M is not MSP the algorithm will answer "No".

-- The last step considers T as an implicit representation of the transtive closure, G_T, of G_M and tests whether G is a subgraph of it. If G is a subgraph of G_T the algorithm gives a "Yes" answer, otherwise it gives a "No" answer.

This algorithm will be described in the next two sections in enough detail to prove that it can be implemented to produce an answer in $O(n+m)$ steps for a digraph with n vertices and m edges. Our description will not follow the flow of control of the algorithm as described above. We

will start by presenting a recognition procedure for the class of MSP digraphs -- the second step of the algorithm -- and then show how to perform the first and third steps, which are more related to what the above description might lead one to believe.

5.3 Recognition and Parsing of MSP Digraphs.

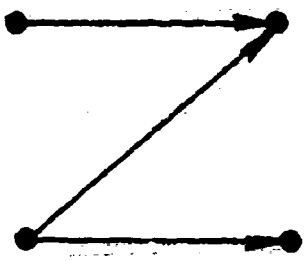
In this section we present an algorithm to solve the following recognition task: given a digraph, G, determine whether G is an MSP digraph, and if it is, obtain a decomposition tree T of G. The algorithm that we present will perform this task in $O(n+m)$ steps for any digraph with n vertices and m edges.

The method that we will use is based on the one-to-one correspondence between TTSP multidigraphs and MSP digraphs given by Lemma 5.3, and Algorithm 4.1 which solved exactly the same task for TTSP multidigraphs. This method can be described as a two step process:

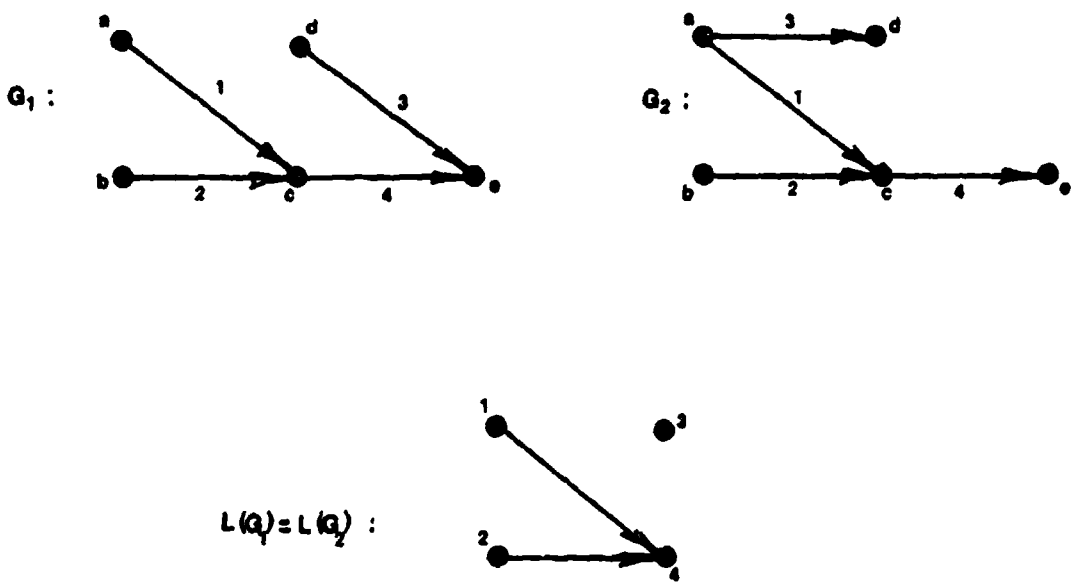
(i) Given G, compute its inverse line digraph $L^{-1}(G)$.

(ii) Use Algorithm 4.1 to determine if $L^{-1}(G)$ is TTSP, and if it is, to obtain a decomposition tree T of $L^{-1}(G)$. If $L^{-1}(G)$ is TTSP we answer "Yes" and output T, otherwise we answer "No".

In principle, Lemma 5.3 would seem to be enough to guarantee the correctness of this process, but there is a problem with the assumption -- made in the first step -- of the existence of an inverse line digraph function. The problem has two aspects: there are digraphs that do not arise as line digraphs, and others that arise as the line digraph of several non-isomorphic digraphs, as shown in Figure 5.6.



(a) A digraph that does not arise as a line-digraph.



(b) Two non-isomorphic digraphs having the same line digraph.

Figure 5.6

The problem of characterizing the digraphs that have line digraph inverses has been extensively studied from a non-algorithmic point of view ([HAR 60], [HEM 72], [KLE 75]) and the problem of computing the line graph of an arbitrary graph has been solved by Lehot [LEH 74]. The approach used by Lehot on undirected graphs is interesting: given a graph G, Lehot computes another graph G_R such that if G is the line graph of any graph, it is the line graph of G_R. He then proceeds to compute the line graph of G_R and test whether it is identical to G. Unfortunately this direct approach works because the inverse line graph -- if it exists -- is unique except in some trivial cases, and does not seem to be useful for our problem where this condition does not hold. Instead we will use a criterion of Harary [HAR 60] to determine whether the input digraph has a line digraph inverse before attempting to compute it.

In the next few paragraphs we will describe Harary's characterization and then describe how we use it to implement the recognition procedure described at the beginning of this section.

Definition 5.3 [Complete Bipartite Composite digraphs]. An acyclic digraph G is Complete Bipartite Composite (CBC) if there exists a set of complete bipartite subgraphs of G : $B_1, B_2, \ldots, B_k$, that we call the bipartite components of G, such that:

(i) each edge of G belongs to exactly one subgraph;

(ii) every vertex v of G, except the sinks, belongs to the head of exactly one subgraph that we will denote by $h(v)$;

(iii) every vertex v of G, except the sources, belongs to the tail of exactly one subgraph that we denote $t(v)$. □

The first part of Harary's characterization is the following:

Lemma 5.4 [HAR 60]. A digraph has an inverse line digraph if and only if it is CBC. □

This lemma solves the problem of the existence of the inverse line digraph, but says nothing about the multiplicity of inverses. The following lemma, due also to Harary, solves this problem:

Lemma 5.5 [HAR 60]. Let G_1 and G_2 be two digraphs such that $L(G_1) = L(G_2)$. The digraphs obtained from G_1 and G_2 by deleting all the sources and sinks are isomorphic. □

Figure 5.7(a) shows what happens after removing all sources and sinks from the two digraphs shown in Figure 5.6(b). Figure 5.7(b) shows the approach that we will use: instead of deleting the sources and sinks we have merged all the sources into a single source and all the sinks into a single sink. The proof of the fact that this operation makes the inverse line digraph unique is a trivial modification of Harary's proof of Lemma 5.5.

We have thus established that any CBC digraph has a unique inverse line digraph having a single source and a single sink. Before we describe how this unique inverse can be computed, we prove some properties of CBC digraphs that we will use later:

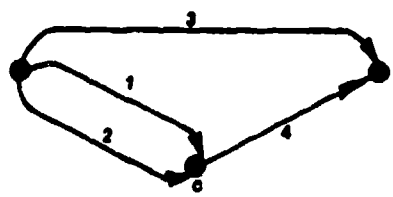
Lemma 5.6.

(i) CBC digraphs are minimal.

(ii) The bipartite components of a CBC digraph are unique.

(iii) Any MSP digraph is CBC.

(a) The unique digraph obtained from the digraphs G_1 and G_2 of Figure 5.6(b) by removing all their sources and sinks.



(b) The unique digraph obtained from the digraphs G_1 and G_2 of Figure 5.6(b) by merging the sources and sinks.

Figure 5.7

Proof. [See Appendix C.] □

The converse of part (iii) of the above lemma is not true: Figure 5.8 shows a CBC digraph that is not MSP.

We now use the results of Lemmas 5.4 and 5.5 to define the inverse line digraph function:

Definition 5.4 [The inverse line digraph function]. Let G be a CBC digraph with bipartite components $B_1, B_2, \ldots, B_k$. The vertex set of $L^{-1}(G)$ -- the inverse line digraph of G -- is $\{B_\alpha, B_1, \ldots, B_k, B_\omega\}$. For each vertex v of G, $L^{-1}(G)$ contains an edge computed as follows:

(i) if v is a source of G, the edge is $(B_\alpha, h(v))$;

(ii) if v is a sink of G, the edge is $(t(v), B_\omega)$;

(iii) if v is a source and a sink, the edge is (B_α, B_ω);

(iv) otherwise the edge is $(t(v), h(v))$. □

The uniqueness of the bipartite components of a CBC digraph (Lemma 5.6) implies that the transformation just defined is a function. The way in which this transformation is an inverse line digraph is given by the following lemma:

Lemma 5.7. $L(L^{-1}(G)) = G$ for any CBC digraph.

Proof. [See Appendix C.] □

Figure 5.9 shows a CBC digraph and its inverse line digraph computed by Definition 5.4.

The refined version of the recognition procedure outlined at the beginning of the section can be described as follows:

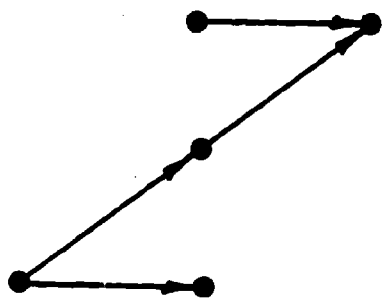
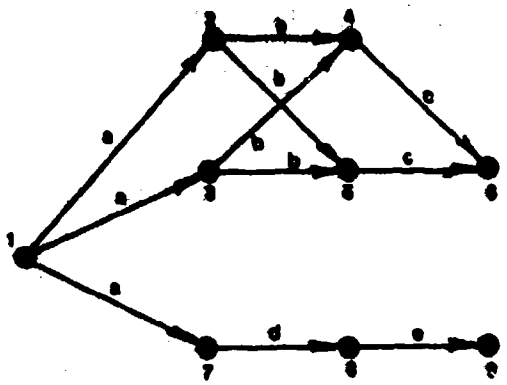


Figure 5.8. A digraph that is CBC and is not MSP.

G:



Vertex	Head	Tail
1	a	Source
2	b	a
3	b	a
4	c	b
5	c	b
6	Sink	c
7	d	a
8	e	d
9	Sink	e

$L^{-1}(G)$:

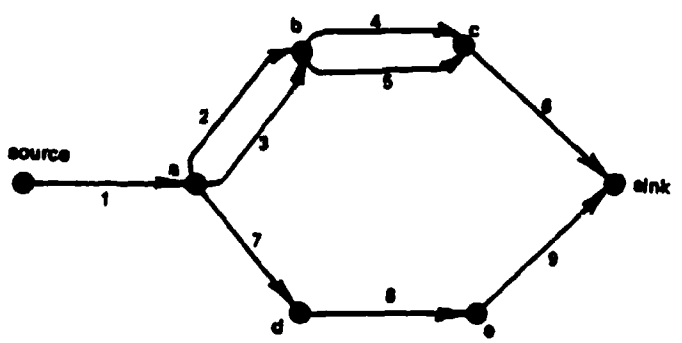


Figure 5.9. A CBC digraph G and its line digraph inverse. The bipartite components of G are identified by letters. $L^{-1}(G)$ is constructed by using the table as an adjacency list, according to Definition 5.4.

Algorithm 5.1 [Recognition of Minimal Series Parallel digraphs].

Input: An acyclic digraph G.

Output: If G is MSP we answer "Yes" and output a binary decomposition tree, T, of G. Otherwise we answer "No".

Step 1: If G is not CBC, answer "No". Otherwise compute $L^{-1}(G)$.

Step 2: If $L^{-1}(G)$ is a TTSP multidigraph, answer "Yes" and output a binary decomposition tree for it. Otherwise answer "No". □

The correctness of this procedure can be derived from Lemmas 5.3, 5.6, and 5.7 as follows:

-- If G is MSP it will also be CBC according to Lemma 5.6 and $L^{-1}(G)$ will be a TTSP multidigraph according to Lemmas 5.3 and 5.7.

-- Because $L(L^{-1}(G)) = G$, a binary decomposition tree of $L^{-1}(G)$ as a TTSP multidigraph will be a binary decomposition tree of G as an MSP digraph according to the discussion given in the previous section.

Let us now consider the problem of implementing Algorithm 5.1 to run in $O(n+m)$ steps for a digraph with n vertices and m edges.

Step 2 can be obviously implemented -- using Algorithm 4.1 -- to run in time proportional to the number of vertices plus the number of edges of $L^{-1}(G)$. Because $L^{-1}(G)$ has an edge for each vertex of G, and at most one vertex for each of the edges of G, Step 2 will run in $O(n+m)$ steps if implemented by Algorithm 4.1.

Let us then consider how to implement Step 1. We proceed as follows: we select an edge (u,v) of G that has not yet been assigned to a bipartite component and mark it as belonging to a new bipartite component B_i.

We now mark all the predecessors of v as belonging to the head of B_i and all the successors of u as belonging to the tail of B_i and then check that there is a complete bipartite subgraph of G with the head and tail just identified. If such a subgraph exists, we mark all its edges as belonging to B_i ; if no such subgraph is found we answer "No" and stop. We then proceed to select a new unmarked edge and repeat the process until all edges have been marked or a "No" answer is generated. While performing this process we answer "No" and stop if we ever attempt to mark an edge as belonging to more than one bipartite component or to mark a vertex as belonging to more than one head or tail.

Because the bipartite components of a CBC digraph are unique, the process just described will identify a new component each time a new edge is selected and processed as explained above. Therefore if this procedure ends without generating a "No" answer, it proves that its input is CBC by identifying the complete bipartite subgraphs that satisfy the conditions of Definition 5.3 and is thus correct.

Once we have decided that G is CBC and identified its bipartite components, the computation of $L^{-1}(G)$ is a trivial application of the rules given by Definition 5.4 so Step 1 can obviously be implemented to run in $O(n+m)$ steps by the above procedure.

This completes our description of the linear time recognition algorithm for the class of MSP digraphs. In the next section we describe how this procedure can be used as part of a linear time recognition algorithm for the class of GSP digraphs.

5.4 Recognition of GSP Digraphs.

We turn now to the central problem of this chapter: given an acyclic digraph G, we want to determine whether G is GSP in a number of steps proportional to the number of vertices plus the number of edges of G.

An approach to this problem is suggested immediately by the relationship between GSP and MSP digraphs (see Definitions 5.1 and 5.2) and the recognition procedure for MSP digraphs just presented. This method can be described as a two step process:

(i) On input G, compute its transitive reduction G_R.

(ii) Use Algorithm 5.1 to determine whether G_R is an MSP digraph. If G_R is MSP answer "Yes" and output a binary decomposition tree of it. Otherwise answer "No".

This process will not only perform the task we want, but wherever it gives a "Yes" answer it will output a binary decomposition tree that represents either the transitive reduction or the transitive closure of its input.

The problem with the process as described resides in the first step: the best known method of computing the transtive reduction of an arbitrary digraph ([AHO 72]) takes $O(n^{\log_2 7})$ steps on a digraph with n vertices. Even worse, the problem is equivalent to computing the transitive closure of an acyclic digraph, so the hope of ever discovering a linear time algorithm for this task is very close to zero.

Fortunately a relatively simple modification of the procedure outlined above can be implemented so it achieves the time bound desired. The modified procedure can be described as follows:

Algorithm 5.2 [Recognition of General Series Parallel digraphs].

Input: An acyclic digraph $G = \langle V, E \rangle$.

Output: If G is GSP, we answer "Yes" and output a binary decomposition tree of the transitive reduction (or transitive closure) of G . Otherwise we answer "No".

Step 1: (Pseudo-transitive reduction.) Partition E into E_M and E_T such that if G is GSP, the digraph $G_M = \langle V, E_M \rangle$ is its transitive reduction. (If G is not GSP, G_M may still be MSP since it will not be in general the transitive reduction of G .)

Step 2: If G_M is not MSP, answer "No". Otherwise compute a binary decomposition tree, T , of G_M as an MSP digraph.

Step 3: Use T as the representation of the transitive closure of G_M and test that all the edges of E_T belong to it. If they do, answer "Yes" and output T , otherwise answer "No". □

The modification introduced in our original description consists of replacing the slow and precise operation of computing the true transitive reduction of the input by two separate processes. We first perform a "quick and dirty" pseudo transitive reduction and then a check of the validity of this pseudo reduction.

We will devote the rest of this chapter to showing how to implement this algorithm to run in $O(n+m)$ steps when its input has n vertices and m edges, but before giving the details we give a proof of the correctness of the algorithm to be implemented.

-- If G is GSP, G_M will be its true transitive reduction and therefore MSP. In this case, the algorithm gives the correct answer since it will give a "Yes" answer and output a binary decomposition tree of G_M.

-- If the algorithm answers "No" in Step 2, G_M is not MSP, and by the condition attached to Step 1, G cannot be GSP. If the algorithm answers "No" in Step 3, it means that G_M is not the true transitive reduction of G which once again implies that G is not GSP.

We therefore conclude that the algorithm is correct.

5.4.1 The Transitive Reduction of GSP Digraphs.

In this section we will describe how to implement Step 1 of Algorithm 5.2 to run in a number of steps that grows linearly with the size of the input digraph.

Remember that what is needed is a procedure that computes the transitive reduction of GSP digraphs and may do anything on a digraph that is not GSP. In particular we do not care if it transforms a digraph that is not GSP into an MSP digraph.

Consider the following functions defined on any acyclic digraph $G = \langle V,E \rangle$ with n vertices and m edges.

The Layer function: $L_G: V \to \{1,2,\ldots,n\}$.

$L_G(v) = 0$ if v is a source, otherwise the length of the longest path from a source of G to v.

The Jump function: J_G: $E \rightarrow \{1,2,\ldots,n\}$.

$J_G((u,v)) = L_G(v) - L_G(u)$.

The Minimum jump function: M_G: $V \rightarrow \{1,2,\ldots,n\}$

$M_G(v) = 0$ if v is a sink of G, otherwise the minimum value of J_G over all edges that leave G .

The values of these functions for a sample digraph are shown in Figure 5.10. Our interest in these functions is due to the following two facts:

Lemma 5.8. Let G be an acyclic digraph and (u,v) a redundant edge of G .

$$M_G(u) < J_G((u,v)) \quad .$$

Proof. [See Appendix C.] □

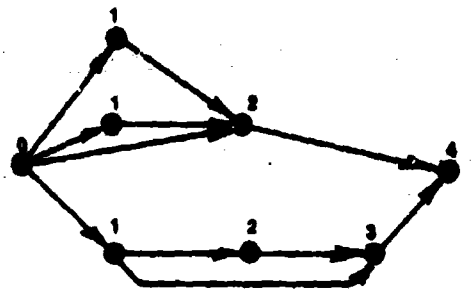
Lemma 5.9. Let G be an MSP digraph. For any edge (u,v) of G ,

$$M_G(u) = J_G((u,v)) \quad .$$

Proof. [See Appendix C.] □

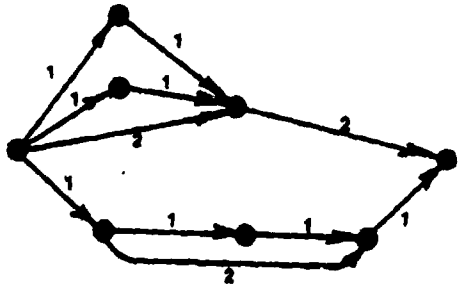
The Jump and Minimum jump functions were defined in terms of the Layer function which in turn was defined by the length of a longest path. Because a path of this type cannot contain redundant edges and the numerical values of L_G , T_G , and M_G are defined by these paths, the three functions are insensitive to the addition or removal of redundant edges. In other words, if u and e are a vertex and an edge of G , and G' is a digraph

G:



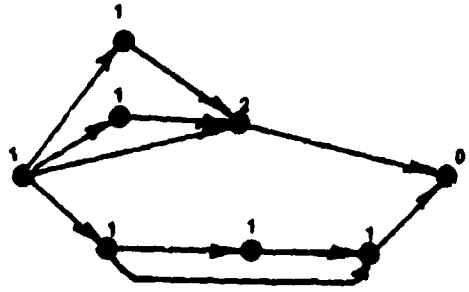
(a) Values of L_G .

G:



(b) Values of J_G computed from the values of L_G above.

G:



(c) Values of M_G computed from the values of J_G above.

Figure 5.10

obtained from G by addition (or removal) of redundant edges $L_G(u) = L_{G'}(u)$, $J_G(e) = J_{G'}(e)$, and $M_G(u) = M_{G'}(u)$.

One can put together the results of Lemmas 5.8 and 5.9 with this property of the functions we have been using to prove the following:

Corollary 5.1. Let G be a GSP digraph and (u,v) one of its edges. The edge (u,v) is redundant in G if and only if $M_G(u) < J_G((u,v))$. □

This corollary tells us that computing the Jump and Minimum Jump functions is enough to perform the transitive reduction of a GSP digraph. Because these two functions can be trivially computed from the values of the Layer function, and these values can be computed by a trivial modification of the topological sort algorithm given by Knuth ([KNU 69]), we can implement Step 1 of Algorithm 5.2 in $O(n+m)$ steps for an acyclic digraph with n vertices and m edges.

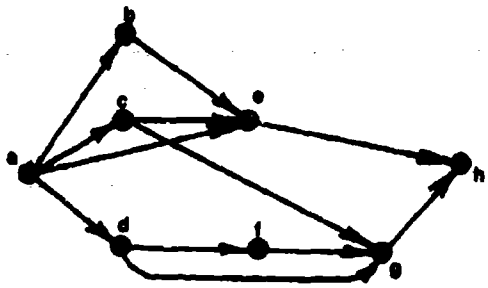
Before we go on to describe the implementation of Step 3 of Algorithm 5.2 it is important to realize that the process just described does not compute the transitive reduction of an arbitrary acyclic digraph. An example of how this method fails on a non-GSP digraph is shown in Figure 5.11.

5.4.2 The Two Dimensionality of GSP Digraphs.

In this section we complete our description of the implementation of Algorithm 5.2 by showing how its last step can be performed in an amount of time proportional to the number of vertices and edges of the input.

The task that we want to perform is the following: we are given a binary decomposition tree T of an MSP digraph G_M, and a set of edges

G :



G′:

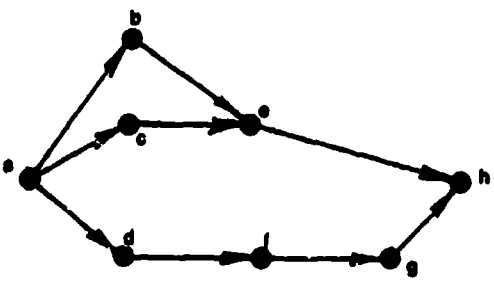


Figure 5.11. If one applies the criterion of Corollary 5.1 to G (which is not GSP) one obtains the digraph G' which is not the transitive reduction of G since edge (c,g) was not redundant.

E_T joining vertices of G_M and we are asked to determine whether the edges of E_T belong to the transitive closure of G_M, in a number of steps proportional to the size of G_M plus the number of edges of E_T.

The method that we will employ is based on some of the properties of the partial order that the edges of a GSP digraph induce on the set of its vertices. We therefore start by studying these properties.

An acyclic digraph can be viewed as defining a partial order on the set of its vertices as follows: for any two vertices u, v we say that $u < v$ if and only if there is a path $u \Rightarrow^* v$ in the digraph. Because this partial order is defined in terms of paths and addition or removal of redundant edges does not create or destroy any paths, the partial order defined by a digraph is the same as the one defined by its transitive closure or its transitive reduction.

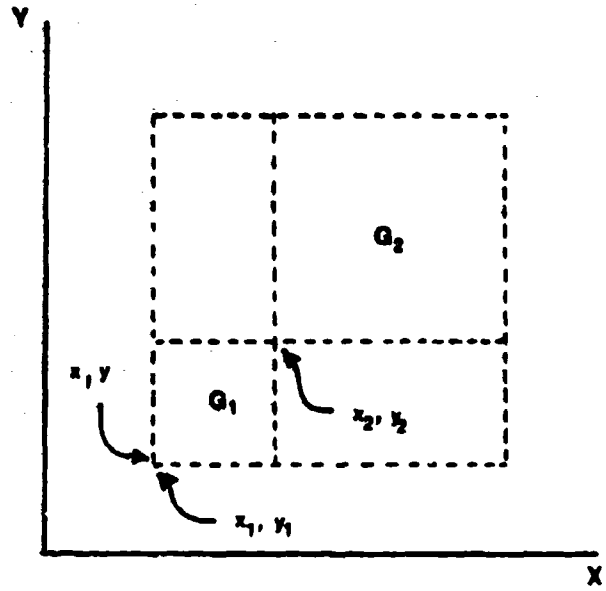
Let us regard a total order t on a set S as a one-to-one mapping $t: S \rightarrow \{1,2,\ldots,\|S\|\}$ and let $t_1, t_2, \ldots, t_k$ be total orders on S. We say that a partial order, $<$, on S is <u>represented</u> by the intersection of $t_1, t_2, \ldots, t_k$ if for any two elements x, y of S, $x < y$ if and only if $t_i(x) < t_i(y)$ for all $1 \leq i \leq k$. The minimum number of total orders needed to represent a partial order in this manner is called the <u>dimension</u> of the partial order.

We will implement Step 3 of Algorithm 5.2 by computing, for any MSP digraph G_M, two total orders whose intersection represents the partial order induced by G_M. Once these orders are computed, to determine whether an edge (u,v) belongs to the transitive closure of G_M we only have to test whether u is ordered with v in both total orders. To make the process of computing these partial orders easier to understand we will use the following geometric interpretation.

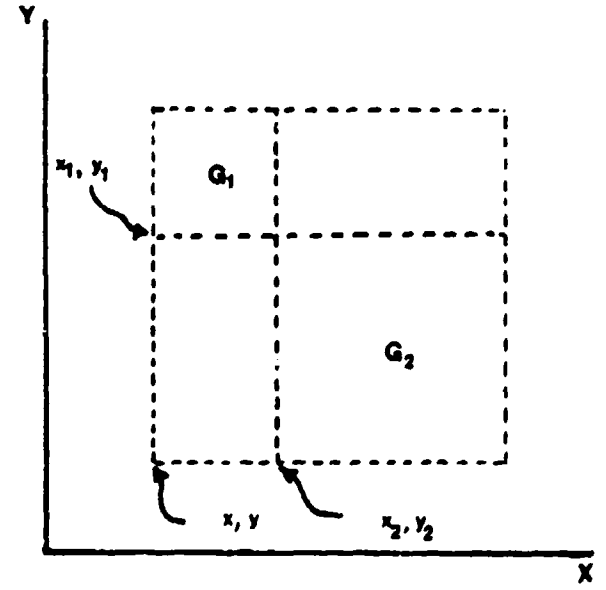
Let $G = \langle V,E \rangle$ be a digraph that induces a two dimensional partial order on V, and let t_1 and t_2 be two total orders whose intersection represents the partial order induced by G. Each vertex $v \in V$ can be assigned coordinates $t_1(v)$ and $t_2(v)$ resulting in an embedding of G in a $\|V\|$ by $\|V\|$ square of the cartesian plane. This embedding is such that for any two vertices u, w of V, there is a path $u \Rightarrow^* w$ in G if and only if the two coordinates of u are pairwise smaller than those of w. Clearly such an embedding can be found for a digraph if and only if the partial order it induces is at most two dimensional, so we have found an intuitive geometric interpretation of the two dimensionality of a digraph.

The method that we will use to embed an MSP digraph on the plane is shown in Figure 5.12. If the vertices of the digraphs G_1 and G_2 involved in a Minimal Series or Parallel computation are placed in the relative positions shown, the coordinates of any two vertices u, v not belonging to the same digraph will satisfy $x_u < x_v$ and $y_u < y_v$ if and only if there is a path $u \Rightarrow^* v$ in the digraph resulting from the composition.

It is not hard to see how this approach can be used recursively to reduce the problem of embedding an MSP digraph with n vertices to n trivial problems involving the embedding of an MSP digraph with a single vertex and no edges at a given point in the plane. The following paragraphs describe how a binary decomposition tree of the MSP digraph to be embedded can be used to perform this task in a straightforward manner.



Minimal Series composition



Parallel composition

Figure 5.12. How to embed the components of a Minimal Series or Parallel composition.

We already know that an n by n square of the plane is enough to embed an MSP digraph with n vertices so all the coordinates of the vertices are integers. Let us now define the position of an MSP digraph on the plane by the coordinates of the lower left corner of the square that contains the vertices of the digraph. Using this convention, we can compute the positions (x_1,y_1) of G_1 and (x_2,y_2) of G_2 in Figure 5.12 if we know (i) the number of vertices n_1 of G_1 and n_2 of G_2, (ii) the position (x,y) of the digraph resulting from the composition of G_1 and G_2, and (iii) the type of composition. These coordinates are related by the following formulae:

Series Composition:

$$x_1 = x \quad ; \quad y_1 = y$$

$$x_2 = x + n_1 \quad ; \quad y_2 = y + n_2$$

Parallel Composition:

$$x_1 = x \quad ; \quad y_1 = y + n_2$$

$$x_2 = x + n_1 \quad ; \quad y_2 = y \quad .$$

Using these formulae and a binary decomposition tree of the MSP digraph to be embedded on the plane we can compute the coordinates of the vertices by the process shown in Figures 5.13 and 5.14.

Figure 5.13 shows an MSP digraph G_0, and a binary decomposition T of G_0. Associated with each node of T we have an integer that tells the size (number of vertices) of the MSP digraph represented by the subtree of T rooted at that node. Note that this value is one for any leaf and the sum of the values of its two children for any internal node; therefore these values can be computed by a single postorder traversal of T.

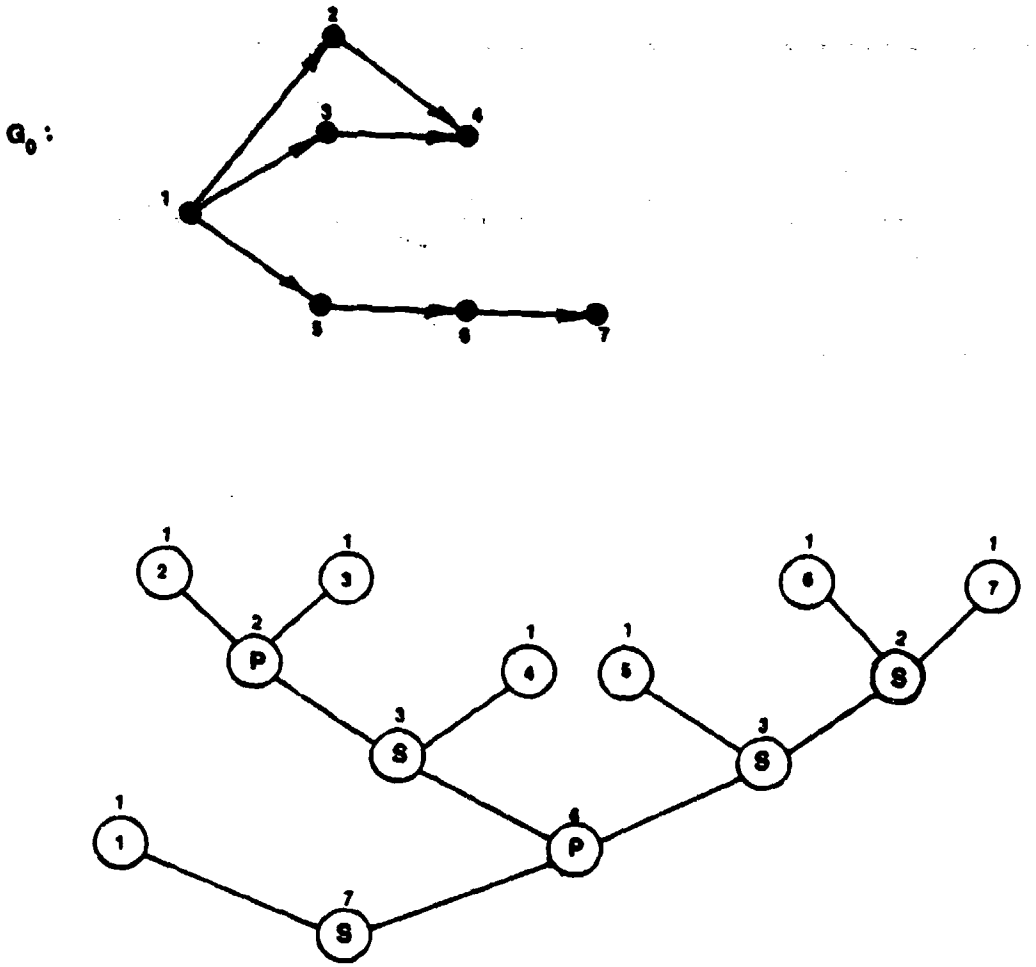
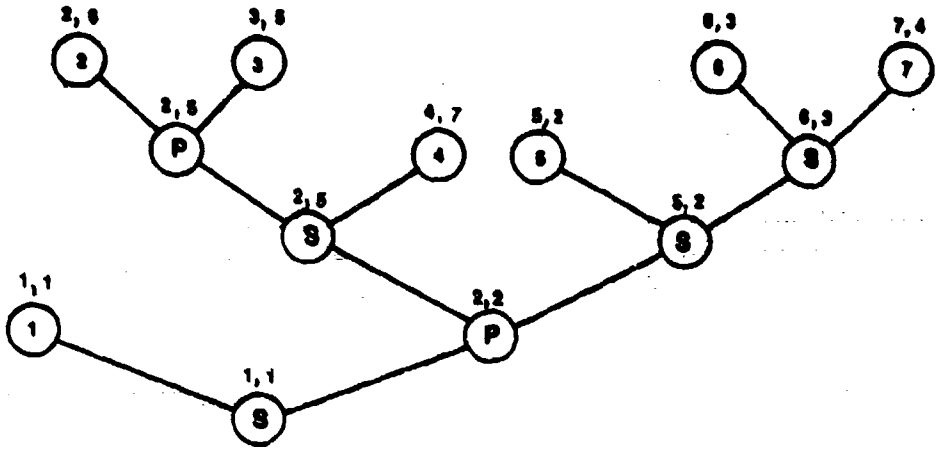


Figure 5.13. An MSP digraph, a binary decomposition tree for it and the sizes of each of the subtrees of the binary composition tree.

T:



Vertex	X	Y
1	1	1
2	2	6
3	3	5
4	4	7
5	5	2
6	6	3
7	7	4

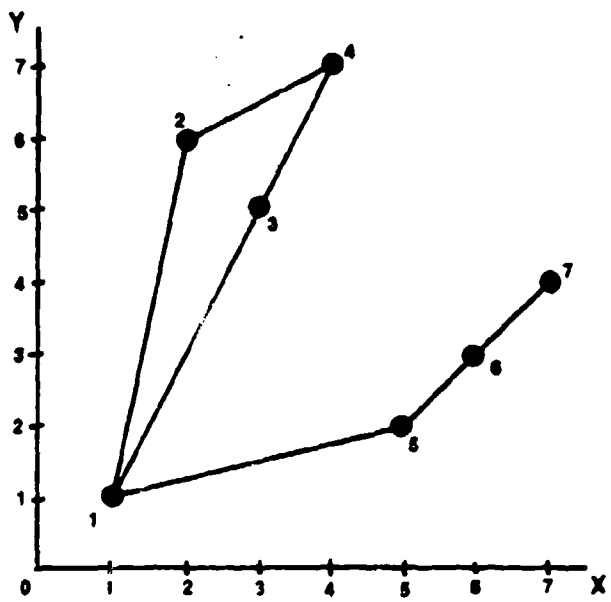


Figure 5.14. Coordinates associated to all vertices of T and resulting embedding of G_0 in the cartesian plane.

In Figure 5.14 we have associated a pair of coordinates with each node of T. The coordinates of any node of T indicate the position of the lower left corner of the square of the cartesian plane that contains the MSP digraph represented by the subtree of T rooted at that node. These pairs of integers have been computed by arbitrarily assigning the pair $(1,1)$ to the root of T, and then traversing T once, from root to leaves, using the formulae given earlier to compute the coordinates of the children of each node visited. Figure 5.14 also shows the embedding of G_0 on the plane that results from this process if one takes the coordinates assigned to each leaf of T as the plane coordinates of the corresponding vertex of G_0.

The processes described in Figures 5.13 and 5.14 can be performed by a single traversal of T each, and will therefore terminate in a number of steps proportional to the number of nodes of T which is in turn proportional to the number of vertices of G_0.

Regardless of the number of steps taken, the fact that any MSP digraph can be embedded in the plane in such a way that for any two of its vertices u, v there is a path $u \Rightarrow^* v$ if and only if the coordinates of u are pairwise smaller than those of v constitutes a proof of the following lemma:

Lemma 5.10. At most two total orders are needed to represent the partial order induced by a GSP digraph on the set of its vertices. □

The converse of this lemma is not true; the digraph of Figure 5.15 induces a two-dimensional order and is not GSP.

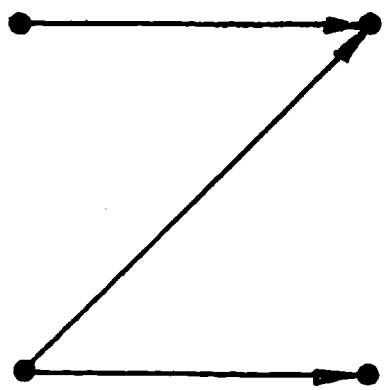


Figure 5.15. The forbidden subgraph for the class of GSP digraphs.

Returning now to Step 3 of Algorithm 5.2, we have described a way to assign coordinates to the vertices of G_M so that determining whether an edge of E_T belongs to the transitive closure of G_M is reduced to comparing the coordinates of its endpoints. Since these coordinates can be computed in a number of steps proportional to the number of vertices of G_M, the discussion of this subsection completes our long proof of the following theorem:

Theorem 5.1. Algorithm 5.2 can be implemented to run in $O(n+m)$ steps on a digraph with n vertices and m edges. □

5.5 Forbidden Subgraph Characterization of GSP Digraphs.

In this section we provide a forbidden subgraph characterization of GSP digraphs based on the digraph of Figure 5.15, which -- for obvious reasons -- will be called N. We will prove the following:

Theorem 5.2. An acyclic digraph G is GSP if and only if it does not contain N as an implicit subgraph, that is, if and only if the transitive closure of G does not contain N as an induced subgraph. □

Of the double implication in the above characterization one of the directions can be proved by a straightforward induction on the number of vertices of the digraph:

Lemma 5.11. Let G be a GSP digraph. G does not contain N as an implicit subgraph.

Proof. [See Appendix C.] □

We will prove the implication in the other direction by describing how to modify Algorithm 5.2 to exhibit the forbidden subgraph whenever it gives a "No" answer. This description will be detailed enough to constitute an algorithm to exhibit the forbidden subgraph in a number of steps proportional to the number of vertices and edges of the input of Algorithm 5.2.

Figure 5.16 shows a flowchart of the GSP recognition algorithm that gives names to the products of its intermediate steps and states where and why the algorithm generates answers. In our discussion throughout the rest of this section we will refer to this figure.

The following lemma plays a central role in our proof:

Lemma 5.12. Let $(u,v) \in E_T$. Either (u,v) is redundant in G or there are edges (u,x) and (y,v) in G such that $J_G((y,v)) = 1$ and $M_G(u) = J_G((u,x))$ and x, y, u, and v are the four vertices of an implicit N subgraph of G.

Proof. [See Appendix C.] □

This lemma implies directly that if Algorithm 5.2 answers "No" in Step 3, G contains N as an implicit subgraph because a "No" answer at that point means that some edge of E_T is not redundant in G. Additionally, it is trivial in this case to exhibit the forbidden subgraph: if (u,v) is the non-redundant edge of E_T, any successor, x, of u and any predecessor, y, of v that satisfy the conditions of the above lemma will form with u and v an implicit N subgraph of G.

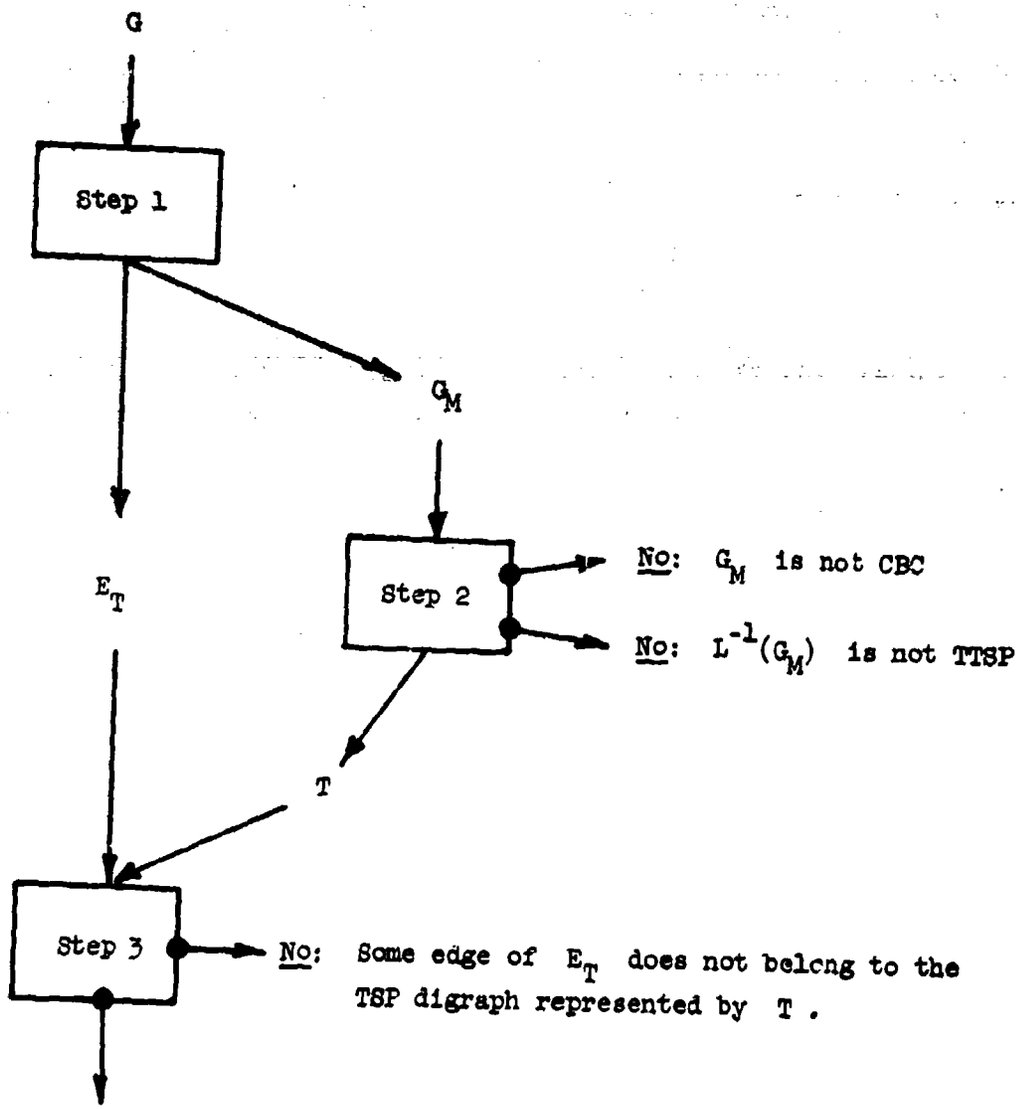


Figure 5.16. Schema of Algorithm 5.2.

Another important consequence of Lemma 5.12 is that if G_M contains N as an implicit subgraph, G will also contain N as an implicit subgraph. We prove this fact by the following argument: if some non-redundant edge of G is deleted to obtain G_M, G contains an implicit N subgraph already; otherwise if all the edges removed from G are redundant there is a path $w \Rightarrow^* z$ in G if and only if there is a path $w \Rightarrow^* z$ in G_M and thus any four vertices that form an implicit N subgraph of G_M will form an implicit N subgraph of G as well.

We will now use this property to complete our argument by proving that if Algorithm 5.2 gives a "No" answer in Step 2 -- either because G_M is not CBC or because $L^{-1}(G)$ is not a TTSP multidigraph -- G_M contains N as an implicit subgraph.

We start by proving that if G_M is not CBC it contains an implicit N subgraph. To prove this fact we examine the procedure (described in Section 5.3) to test whether a digraph is CBC. This procedure was the following:

(a) Select an unmarked edge (u,v) of G_M.

(b) Identify two sets $H = \{x \mid (x,v) \in G_M\}$ and $T = \{x \mid (u,x) \in G_M\}$.

(c) Test whether there is a complete bipartite subgraph of G_M with head H and tail T; if such a subgraph exists, mark all its edges, otherwise answer "No".

(d) Repeat (a), (b), and (c) until either all edges are marked or until a "No" answer is generated.

Step (c) can be performed as follows (taking advantage of the knowledge that G_M is minimal and contains no multiple edges):

(i) test that every edge that leaves a vertex in H enters a vertex of T ;

(ii) test that every edge that enters a vertex in T leaves a vertex of H ;

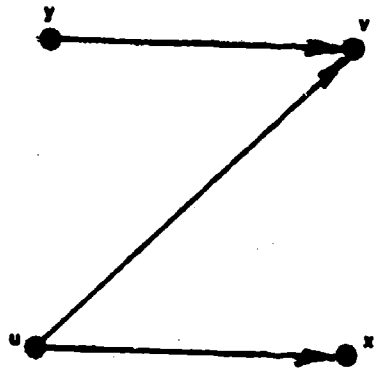
(iii) test that there are exactly $\|H\| \times \|T\|$ edges (x,y) such that $x \in H$, and $y \in T$.

Suppose that (i) is not the case and there is an edge (x,y) such that $x \in H$ and $y \notin T$. Then the vertices $u, v, x,$ and y form an <u>induced</u> N subgraph of G_M since (u,v) , (x,y) , and (x,v) are edges of G_M and (u,y) is not (or y would belong to T). We can argue in the same manner to show that if (ii) is not the case we can exhibit an <u>induced</u> N subgraph of G_M .

Let us now consider the situation when (iii) is not the case. Let k be the number of edges counted in (iii). Because G_M is a digraph, $k \leq \|H\| \times \|T\|$ and since (iii) is not the case it must be that $k < \|H\| \times \|T\|$. There must therefore exist a pair of vertices $x \in H$ and $y \in T$ such that $(x,y) \notin G_M$. Once again x, y, u and v will form an <u>induced</u> N subgraph of G_M because edges (u,v) , (x,v) , and (u,y) are all in G_M while (x,y) is not.

This argument proves that if G_M is not CBC it contains an <u>induced</u> N subgraph. To prove that the four vertices of this induced subgraph are the vertices of an <u>implicit</u> N subgraph of G_M we argue as follows. Let the vertices x, y, u and v form an induced N subgraph of G_M as shown in Figure 5.17(a). If there is a path $x \Rightarrow^* y$ in G_M , the edge (u,v) would be redundant implying that some redundant edge of G

(a)



(b)

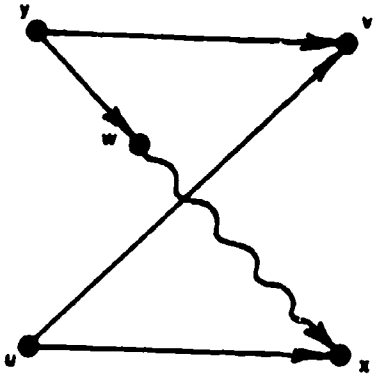


Figure 5.17

was not removed in Step 1 of Algorithm 5.2 which contradicts Lemma 5.8. If there is a path $y \Rightarrow^* x$ as shown in Figure 5.17(b), we have:

-- $L_G(w) < L_G(x)$ because the values of L_G increase along any path of G_M.

-- $L_G(v) = L_G(w)$ or one of (y,v), (y,w) would have been deleted in Step 1.

-- $L_G(v) = L_G(x)$ or one of (u,x), (u,v) would have been deleted in Step 1.

These three facts are clearly contradictory so we conclude that there is no path $y \Rightarrow^* x$.

We therefore conclude that if G_M is not CBC it contains N as an implicit subgraph.

Let us consider the case when the algorithm answers "No" because $L^{-1}(G_M)$ is not a TTSP multidigraph although G_M is CBC.

Because $L^{-1}(G_M)$ is computed by Definition 5.4, it will be acyclic and have a single source and a single sink. Thus, according to Lemma 4.4, $L^{-1}(G_M)$ contains the Wheatstone bridge as an embedded subgraph since it is not TTSP.

The gist of the remaining argument is contained in Figures 5.18 and 5.19. In Figure 5.18 we show the results of computing the line digraph of a Wheatstone bridge and of a general path. Figure 5.19 puts these two facts together to show the result of computing the line digraph of a generalized Wheatstone bridge: a generalized N digraph.

Let $L^{-1}(G_M)$ contain an embedded Wheatstone bridge -- the pattern of Figure 5.19(a) -- and be acyclic. Its line digraph, G_M, will contain an implicit N subgraph, because it will contain the pattern

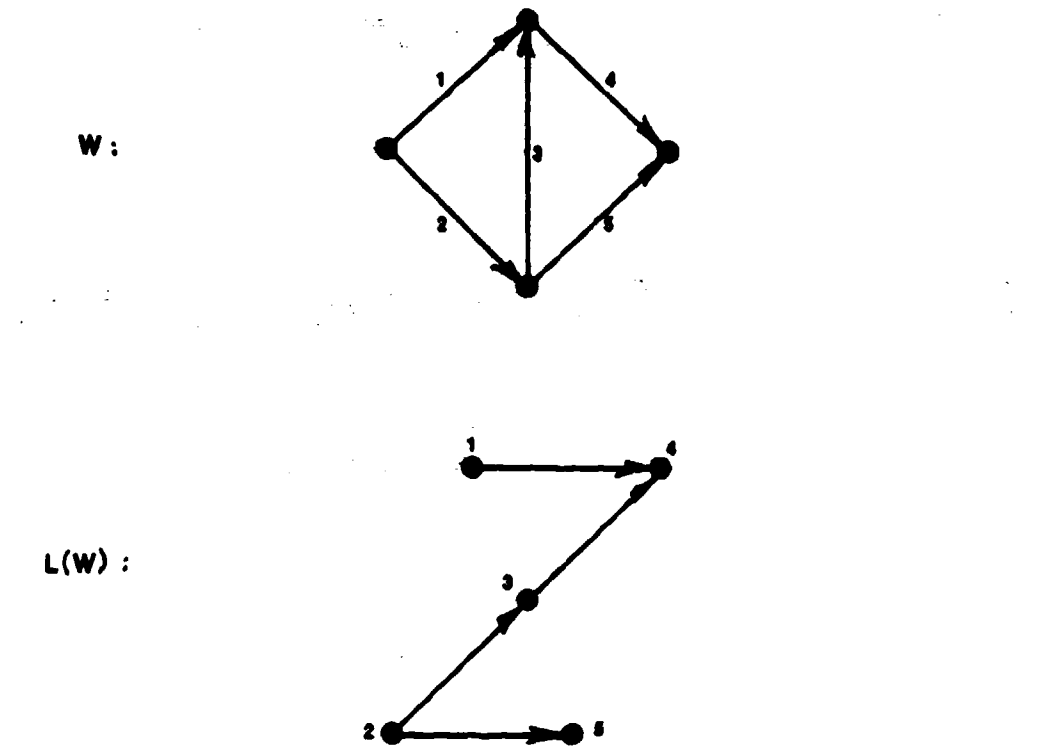
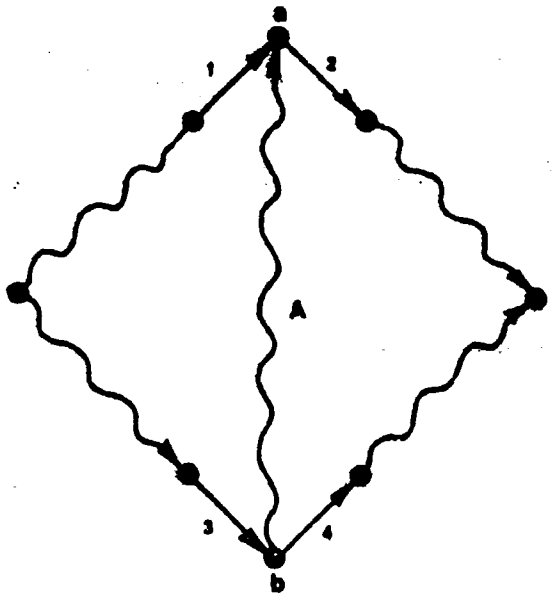


Figure 5.18. The line digraphs of two particular digraphs.

(a) G_1:



(b) $L(G_1)$:

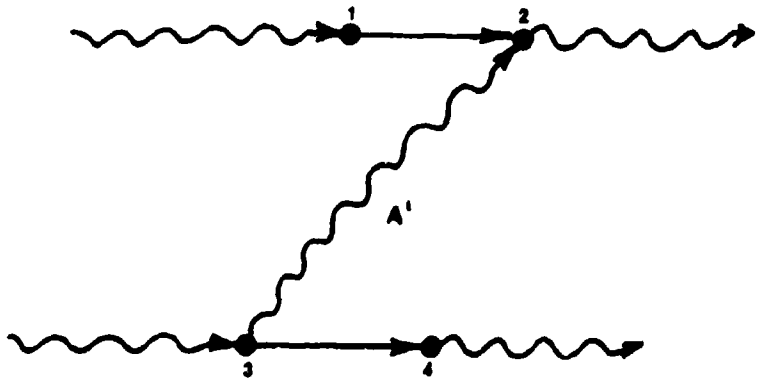


Figure 5.19. The line digraph of a generalized Wheatstone bridge. (The path labelled A' in $L(G_1)$ arises as the line digraph of the path labelled A in G_1.)

of Figure 5.19(b) and existence of paths $1 \Rightarrow^* 4$ or $4 \Rightarrow^* 1$ in G_M would imply the existence of a cycle in $L^{-1}(G_M)$. Furthermore by finding the edges labelled 1, 2, 3, and 4 of the embedded Wheatstone bridge of $L^{-1}(G_M)$ one knows the four vertices of the implicit N subgraph of G_M. Thus the procedure to exhibit the Wheatstone bridge described in Section 4.5 can be used directly to exhibit the implicit N subgraph of G_M in this case.

We therefore conclude that if Algorithm 5.2 answers "No" in Step 2, G_M contains N as an implicit subgraph. Because we proved that if a "No" answer was produced in Step 3, G contained an implicit N subgraph and also that if G_M contains an implicity N subgraph so does G, this completes the proof of Theorem 5.2. However, we stated that our proof would constitute an algorithm to exhibit the forbidden subgraph of G, and we have not yet fulfilled this promise. We have shown how (i) to exhibit the forbidden subgraph of G when the answer "No" is produced in Step 3 and (ii) how to exhibit an implicit N subgraph of G_M when the "No" answer is generated in Step 2. In both cases the procedures described would work in a number of steps proportional to the number of vertices and edges of G. We will end this section by describing how to exhibit an implicit N subgraph of G when given an implicit N subgraph of G_M in a number of steps proportional to the size of G.

Let us assume that G has n vertices and m edges and that the implicit N subgraph of G_M is the digraph of Figure 5.20. First we test whether the paths $x_1 \Rightarrow^* x_4$ or $x_4 \Rightarrow^* x_1$ are present in G, a task that can be performed in at most O(m) steps. Clearly both paths

Figure 5.20. The induced N subgraph of G_M.

cannot be present or G would contain a cycle and if neither is present x_1, x_2, x_3, and x_4 are the vertices of an implicit N subgraph of G, therefore we have only one more case to consider. Let one of the two paths, p, exist in G; because neither path exists in G_M at least one non-redundant edge of p belongs to E_T. We complete our description by showing how one such edge can be identified in at most $O(n+m)$ steps.

Let $v_1, v_2, \ldots, v_k$ be the vertices on p. The values of the layer function increase along any path in G, therefore $L_G(v_i) < L_G(v_{i+1})$ for any two consecutive vertices of p. Let now (v_j, v_{j+1}) be an edge of p that belongs to E_T. We can determine whether this edge is redundant in G by a limited depth-first search that starts at v_j and visits only vertices w for which $L_G(w) \leq L_G(v_{j+1})$. By limiting the search in this way we can test whether each of the edges of p that are in E_T are redundant without visiting any vertex more than twice. We can therefore find a non-redundant edge of p that belongs to E_T in at most $O(n+m)$ steps. Once this edge e has been found, we can exhibit an implicit N subgraph of G by inspecting the vertices adjacent to the endpoints of e looking for two that satisfy the conditions stated in Lemma 5.12.

5.6 Consequences of the GSP Recognition Algorithm.

We end this section by considering briefly a number of problems with a common characteristic: the results presented in this chapter give new insights on how to solve them on GSP digraphs, resulting in some cases in

algorithms that are more efficient than the best algorithms known to solve the problem on an arbitrary digraph.

Transitive Reduction.

Computing the transitive reduction of an acyclic digraph is equivalent to computing its transitive closure. The best known algorithm to perform this task on an arbitrary digraph with n vertices takes $O(n^{\log_2 7})$ steps ([AHO 72]).

In Section 5.4.1 we showed however how the transitive reduction of a GSP digraph can be computed in a number of steps proportional to the number of its vertices and edges.

Transitive Closure.

When we give a GSP digraph as input to Algorithm 5.2 we get not only a "Yes" output but a binary decomposition tree that represents its transitive closure. In Section 5.4.2 we showed how we can use this decomposition tree to compute an implicit description of the transitive closure of the graph so questions like "does edge e belong to the transitive closure?" could be answered in a constant number of steps. Using this approach one can produce the transitive closure of a GSP digraph G in $O(m^*)$ steps, where m^* is the number of edges in the transitive closure of G.

Isomorphism.

In Section 5.2 we described how an MSP or TSP digraph can be represented uniquely by a decomposition tree. Because of the formal identity of the decomposition trees for TTSP multidigraphs and MSP or TSP digraphs, all we said about deciding isomorphism of TTSP multidigraphs

using their decomposition trees in Section 4.6 applies directly to MSP and TSP digraphs as well. In particular we can decide whether two MSP or TSP digraphs are isomorphic in $O(n+m)$ steps: we need $O(n+m)$ steps to compute their decomposition trees using Algorithm 5.2 and and $O(n)$ steps to decide whether the trees are isomorphic.

It is important to realize however that the isomorphism problem for GSP digraphs is equivalent to the problem of isomorphism for arbitrary graphs. This can be proved easily by the construction of Figure 5.21 due to Lawler and Tarjan. In that figure we show a GSP digraph computed from an arbitrary graph $G = \langle V, E \rangle$ as follows:

(i) The vertex set of the GSP digraph is $\{\$\} \cup V \cup E$.

(ii) There is an edge $(v,\$)$ for each $v \in V$.

(iii) There is an edge $(\$,e)$ for each $e \in E$.

(iv) For each $e = (u,v)$ belonging to E, there are edges (u,e) and (v,e).

This digraph is clearly GSP, and one can prove easily that two graphs are isomorphic if and only if the GSP digraphs computed from them in this way are isomorphic. Therefore if we could solve the isomorphism question for GSP digraphs in polynomial time we would be able to solve the isomorphism of arbitrary graphs in polynomial time as well.

Subgraph Isomorphism.

The subgraph isomorphism problem for MSP or TSP digraphs remains an open question in the same way that the problem remains open for TTSP digraphs: there is some hope of finding an efficient algorithm to solve the problem based on Matula's algorithm for subtree isomorphism but we

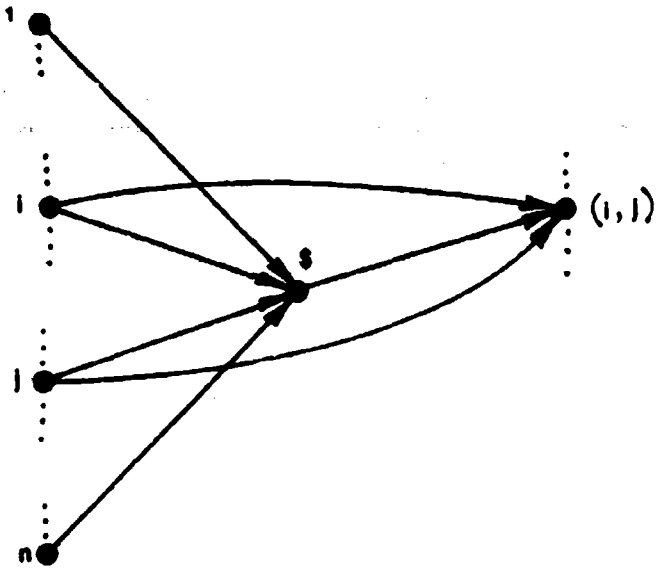


Figure 5.21

don't know how to do it. For GSP digraphs, the construction explained earlier can be used directly to prove that the problem is equivalent to the subgraph isomorphism problem for arbitrary graphs and is therefore NP-complete.

Chapter 6. Flowcharts.

6.1 Introduction.

This chapter discusses how to analyze the flow of control of a program using some of the techniques introduced in earlier chapters.

For the purpose of analyzing its flow of control, a program P is customarily represented by a digraph -- called its control flow graph -- computed as follows:

-- for each group of statements of P that are always executed sequentially, there is a vertex in the control flow graph of P ; and
-- for each possible transfer of control between two such groups in P there is an edge that joins the corresponding vertices of the control flow graph of P .

The problem that we consider can now be described in the words of Kennedy [KEN 71]: "... we would like to know which definitions can affect computations at a given point in the control flow graph and which uses can be affected by computations at a given point. Once we have this information, we can do things like common subexpression elimination, code motion, constant propagation and register allocation."

Because of its obvious application to the design of compilers, the problem of flow analysis has been extensively studied in recent years ([ALL 70], [COC 70], [GRA 76], [HEC 72], [HEC 74], [HEC 77], [KAS 75], [KEN 71], [ROS 77], [TAR 74]).

A method suggested by Rosen ([ROS 77]) to solve some of the data flow problems just mentioned uses information generated during the syntax analysis phase of the compilation of the program to be analyzed. This

approach is particularly well suited to the analysis of goto-less programs written in languages that have a well structured set of control statements, but it is not the technique most commonly used. Normally the problem is tackled by representing some intermediate form of the program being compiled by its control flow graph, analyzing this digraph, and then using the information obtained to generate code for the target machine.

The techniques that we discuss in the remainder of this chapter are based on this last approach and all of them work in two clearly distinguishable stages:

-- first structural information about the control flow graph is obtained by regarding it as an abstract digraph, and then
-- this structural information is combined with information about the computation associated with the vertices of the digraph to solve the data flow questions identified above.

We will concern ourselves exclusively with the first of these two stages: the extraction of structural information from a control flow graph.

The "classical" approach to the structural analysis of control flow graphs is the <u>interval analysis</u> technique introduced by Cocke and Allen ([COC 70], [ALL 70]). This technique can be described as follows.

<u>A flowgraph</u> $G = \langle V, E \rangle$ is a directed graph with a distinguished vertex v (called the <u>start</u> vertex) such that for every vertex $u \in V$ there is a path $v \rightarrow^* u$ in G. An <u>interval</u> can be defined as a maximal subgraph having a single entry, that is, every vertex of the interval has all its predecessors in the interval or it is the entry vertex. The interval analysis of a flowgraph is performed by identifying its intervals,

replacing each of them by a single vertex and repeating the process on the flowgraph obtained, until no more replacements can be carried out. An example of this process is shown in Figure 6.1.

Several efficient algorithms are known to perform the interval analysis of a flowgraph ([HOP 72], [TAR 74]) and to use the information obtained to solve data flow problems ([KEN 71], [GRA 76], [TAR 75]). The fastest analysis algorithm ([TAR 74]) runs in $O(m\,\alpha(n,m))$ [*] steps on a flowgraph with n vertices and m edges.

Interval analysis is a very powerful and general method of parsing programs, but as a result of these characteristics, both the generation of the parse of a flowgraph and the extraction of data flow information from this parse are rather complex processes. The basic aim of the parsing method described in the next few paragraphs -- called <u>prime subhammock parsing</u> -- is to simplify these processes at the expense of some power and generality.

A <u>hammock</u> is a strongly connected digraph with two distinguished vertices α and ω and a distinguished edge (ω,α). The vertices α and ω are called respectively the <u>start</u> and <u>finish</u> of the hammock, and the edge (ω,α) is called the <u>return edge</u>. A <u>subhammock</u> is a subgraph S of a hammock H that does not include the return edge of H, that has exactly two boundary vertices x, y, one being an entry and the other an exit[**] and such that

---

[*] The α function is an inverse of Ackermann's function that grows extremely slowly and can be considered less than four for all practical purposes.

[**] We will discuss the definitions of entry and exit later since the theory that we will present depends somewhat on how these definitions are chosen. For the moment let us say that we will try to approximate the following intuitive idea: an entry (exit) of a subgraph S is a boundary vertex v of S such that every path $\alpha \Rightarrow^* x$ $(x \Rightarrow^* \omega)$ in which $x \in S$ includes the vertex v.

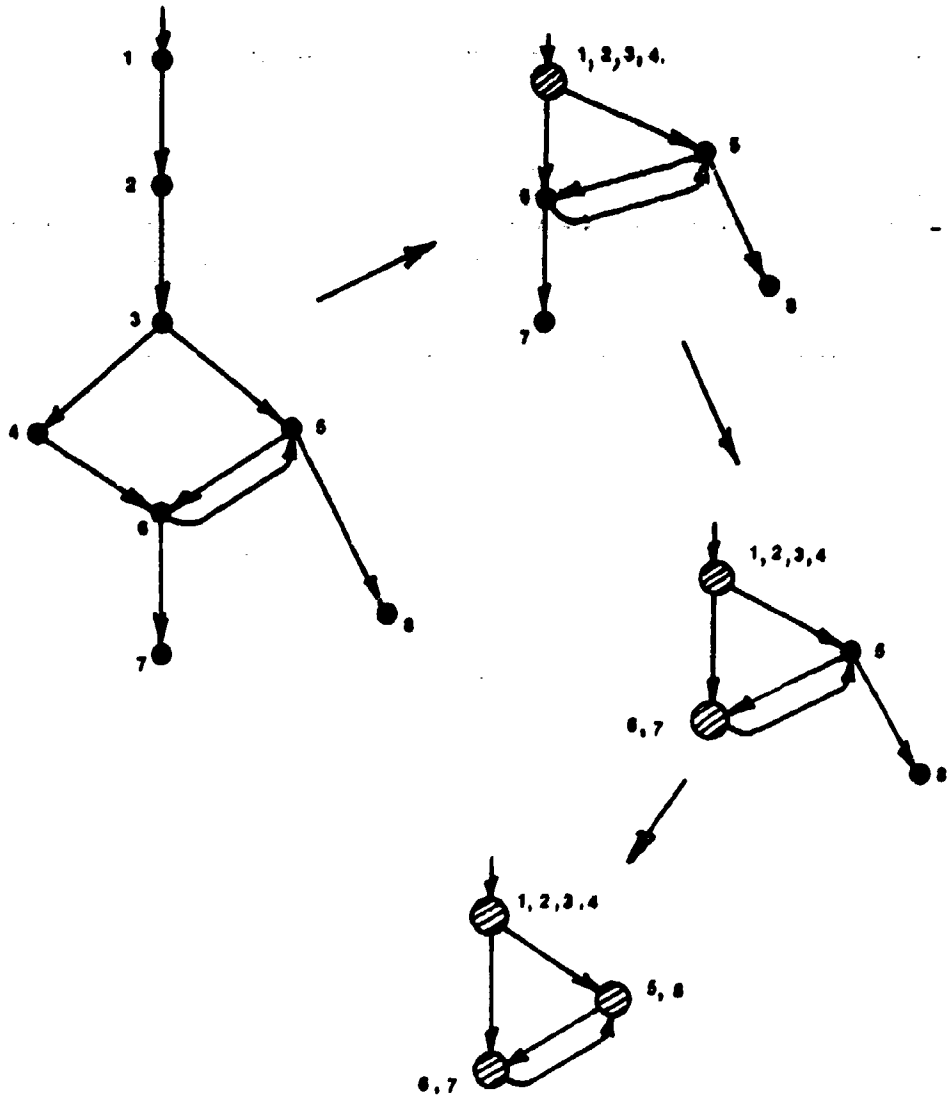


Figure 6.1. Interval analysis of a flowgraph
(start vertex indicated by an edge "from nowhere")

(i) either $S-\{x,y\}$ includes at least one vertex and is connected; or

(ii) S is the maximal subgraph of H with x and y as boundaries not including the return edge.

A subhammock is <u>non-trivial</u> if it includes at least two edges, and it is <u>prime</u> if it is non-trivial and does not properly contain any non-trivial subhammock.

The prime subhammock parse of a hammock H is performed by identifying a prime subhammock of H, replacing it by a single edge going from its entry to its exit, and repeating the process on the resulting digraph until no more replacements can be carried out. An example of this process is shown in Figure 6.2. Clearly, this process would not make a lot of sense unless the digraph obtained from a hammock by the replacement operation just described is a hammock as well. It is therefore necessary that the definitions of entry and exit used be such that this condition can be guaranteed.

Let H be a hammock and let $N(H)$ denote its undirected version. The following fact is the basis of our algorithm to compute prime subhammock parses:

<u>Lemma 6.1</u>. Let H be a hammock such that $N(H)$ is biconnected and let S be a non-trivial subhammock of H. Either S includes every edge of H except the return edge or the entry and exit vertices of S are a separation pair of $N(H)$.

<u>Proof</u>. [See Appendix C.] □

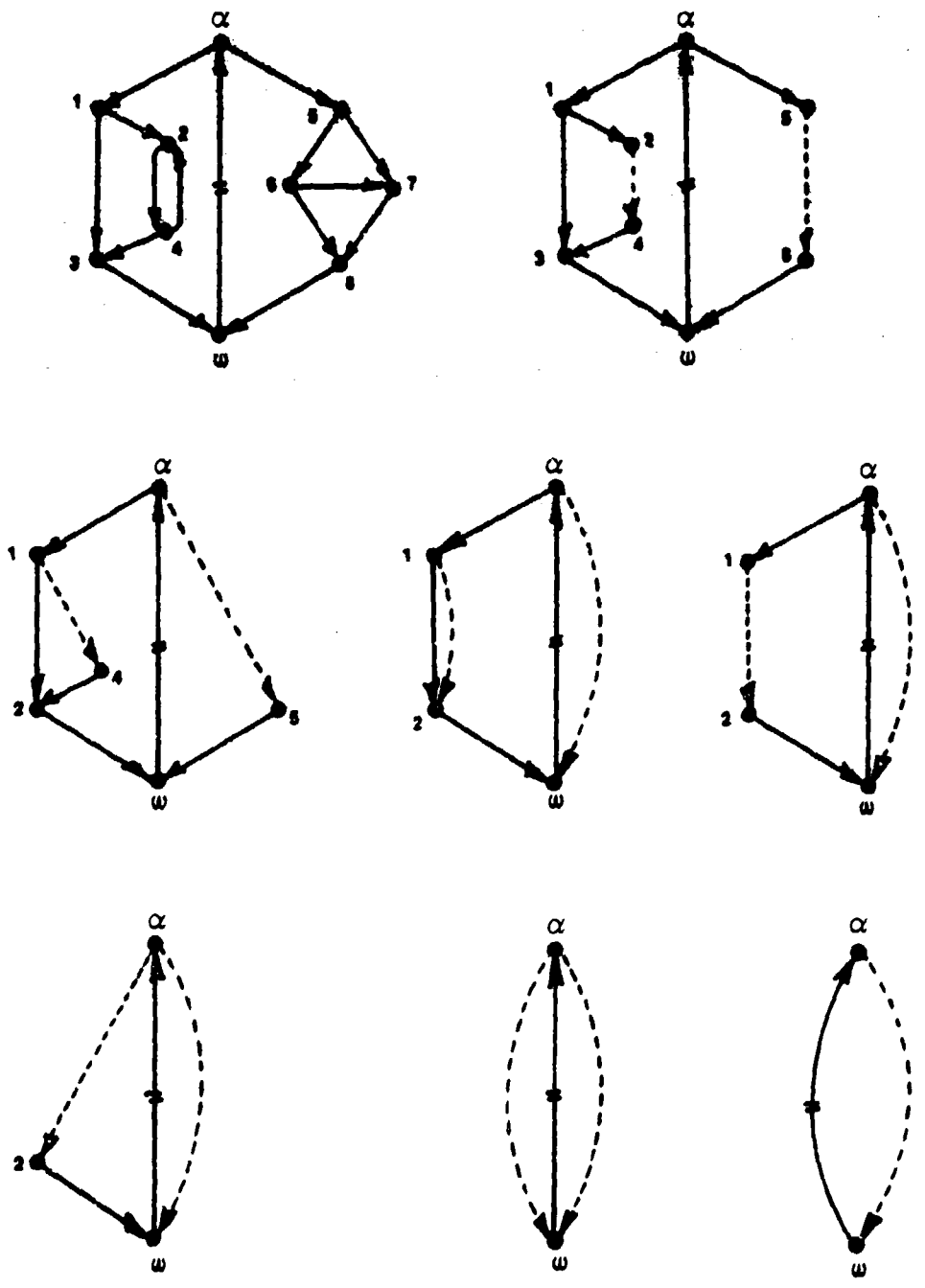


Figure 6.2. Analysis of a Hammock by prime subhammocks. (The return edge has been marked by a double cross.)

This relationship between entry - exit pairs of a hammock H and the separation pairs of $N(H)$ can be used to compute prime subhammock parses as follows:

-- Given a hammock H, compute $N(H)$.

-- Use the triconnected components algorithm to find the separation pairs of $N(H)$.

-- Examine these separation pairs to determine the entry - exit pairs of H.

In the next section we will describe in detail an efficient algorithm to perform the prime subhammock parse of a general hammock based on the outline given above. As we shall see, in the most general case this algorithm becomes quite complicated. This complexity however can be greatly reduced when the hammocks being parsed satisfy certain conditions. For this reason, following the section describing the general case, we have included two others discussing two classes of hammocks that can be parsed by simplified versions of the general algorithm: proper programs and structured programs.

Our discussion concerning proper programs includes a justification of our claim about the convenience of the structural information produced by our parsing algorithm. Proper programs were introduced by Linger and Mills in connection with a method of solving data flow problems on them; we will show that the prime subhammock parse of a proper program generates exactly the structural information needed to perform the data flow analysis suggested by these authors.

6.2 Parsing General Hammocks.

In this section we describe an algorithm to perform the prime subhammock parse of a general hammock H in a number of steps proportional to the number of vertices and edges of H.

Our algorithm will accept as input a general hammock H with n vertices and m edges and output a tree structure T_p that represents all possible prime subhammock parses of H. The structure T_p will be computed as follows:

-- We use the triconnected components algorithm to obtain the TCG T of $N(H)$, and then

-- use T to examine the separation pairs of $N(H)$ and determine which ones are entry - exit pairs of H. This information is then incorporated into T to produce the desired structure T_p.

If we assume that $N(H)$ is a biconnected multigraph, it is clear that the first step of the above process can be performed in $O(n+m)$ steps, so let us consider the second step.

We will perform the examination of the separation pairs of $N(H)$ as follows. We consider T as a rooted tree, its root being the vertex associated to the triconnected component of $N(H)$ that includes the return edge. For each vertex v of T we will consider one separation pair of $N(H)$ corresponding to the endpoints of the virtual edge that the triconnected component associated with v shares with the triconnected component associated with the parent of v in T. This separation pair will be examined to see whether it is an entry - exit pair of the subgraph of H that includes all the edges of H belonging to triconnected components

of $N(H)$ associated to vertices of the subtree of T rooted at v. The tree T_p is computed as follows. For each separation pair of $N(H)$ examined that is not an entry - exit pair of H, we carry out a merge operation among components of $N(H)$ that eliminates the virtual edge joining the separation pair. Resulting from this process will be a set of biconnected multigraphs containing all the edges of $N(H)$ (each edge in one graph) and virtual edges (each shared by two graphs) such that:

(i) $N(H)$ can be obtained from this set by merging operations that eliminate all the virtual edges.

(ii) The endpoints of each virtual edge of this set of graphs are an entry -exit pair of H.

This set can be represented as a tree, in the same way that the triconnected component set was represented by the TCG. This tree T_p is the structure that we will use to represent all possible prime subhammock parses of H.

Two very basic points should be noted about the process just described:

(i) for each separation pair considered we <u>do not</u> examine all possible partitions of the edge set of $N(H)$ determined by that pair (for instance, a bond with k edges would determine $O(2^k)$ such partitions),

(ii) we <u>did not</u> examine explicitly every separation pair of $N(H)$ (for instance, a polygon with k vertices has $O(k^2)$ separation pairs).

We will consider the reasons for these decisions in a moment. Before doing so however, let us examine a more basic question that we have to resolve if we are to keep any hope of performing the simplified process described above in $O(n+m)$ steps.

The problem is that, even though we do not consider all possible separation pairs of $N(H)$, we may still be examining $O(n)$ separation pairs in the process described; therefore if we want to stay within $O(n+m)$ steps we cannot examine each pair for very long. A sufficient condition to guarantee that all pairs that our simplified process considers can be examined in $O(n+m)$ steps is the following: we require our definitions of entry and exit to be such that one can determine whether a vertex v is an entry (exit) of a subgraph S by examining only the edges incident to v asking about them only whether they belong to S or not.

Later in this section we will provide definitions of entry and exit that satisfy these conditions and we will consider their relationship with the intuitive ideas of entry and exit discussed in the previous section.

Let us now return to the problems presented by the multiplicity of edge partitions defined by bonds and the multiplicity of separation pairs defined by a polygon.

The key to our solution of the first problem is our definition of subhammocks. Figure 6.3 shows a hammock H, $N(H)$, and the triconnected components of $N(H)$. According to the procedure just described we will test the separation pair u, v -- at different points -- as to whether it is:

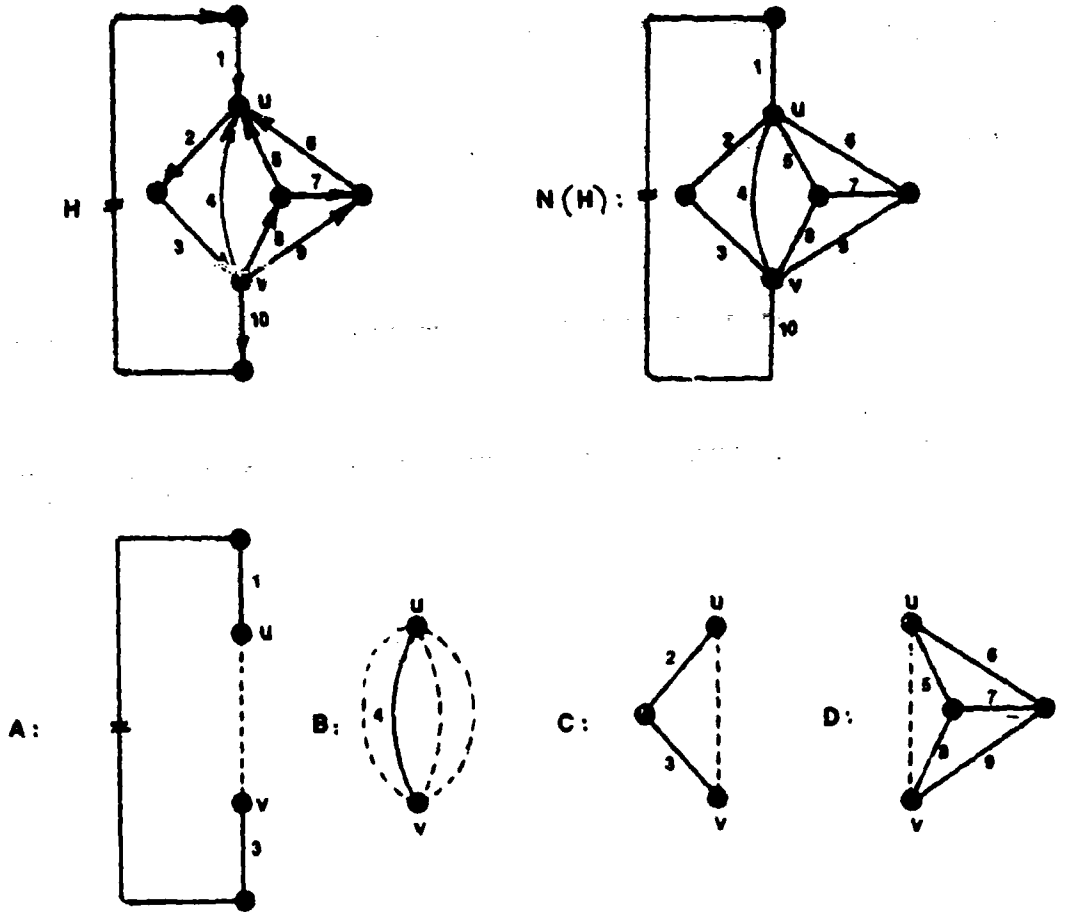


Figure 6.3. A hammock H , its undirected version N(H) , and the triconnected pieces of N(H) .

(i) an entry - exit pair of the subgraph of H containing the edges of C ; or

(ii) an entry - exit pair of the subgraph of H containing the edges of D ; or

(iii) an entry - exit pair of the subgraph of H containing the edges of the subgraph of H including the edges of B , C , and D .

These tests correspond exactly to the non-trivial connected subgraphs when u and v are removed (cases (i) and (ii)) and to the maximal subgraph of H with just u and v as boundary vertices (case (iii)). It is not hard to see how this argument can be generalized to show that we are considering all possible partitions that may result in subhammocks. (For instance we do not consider u , v as a possible entry -exit pair of the subgraph of H including the edges of C and D , since that graph is not connected when u and v are removed.)

It must be apparent that our definition of subhammock was somewhat "ad hoc", however we must say in our defense that it is not much more so than the definitions used by other authors. As an example Kas'janov ([KAS 75]) defines subhammocks as including their entry vertex but not including their exit, therefore eliminating the problem being considered here. Additionally we feel that a measure of reasonableness is given by the fact that in some restricted cases (proper programs) our parsing method coincides with those defined by other authors that look at the problem from a very different perspective. Because there is not much that one can prove in general about the relative merits of definitions of this type we will not attempt to defend our choice any further.

The problem presented by the multiple separation pairs determined by a polygon is similar to the problem we encountered parsing TT networks: there are $O(k^2)$ ways of reducing a polygon with k vertices to a double bond, but all of them can be represented concisely by giving the sequence of vertices that one encounters when circling the polygon in either direction. In our present context we resolve the problem in a similar way as follows.

We will assume that the definitions of entries and exits are such that if a vertex v_i (as depicted in Figure 6.4) is an entry (exit) of the subgraph G_i, it would automatically be an exit (entry) of the subgraph G_{i+1}. With this assumption, all possible entry-exit pairs determined by a polygon like the one of Figure 6.4, can be described by giving the sequence of vertices that are entries - exits of the subgraphs $G_0, G_1, \ldots, G_k$ in the order in which they are encountered as one goes from v_0 to v_k (or from v_k to v_0).

The problem of explicitly exhibiting all the entry exit pairs of a hammock has been studied by Kas'janov ([KAS 75]) who gives an algorithm based on depth-first search that runs in $O(nm)$ steps for a hammock with n vertices and m edges. This bound is improved in our case by not producing the entry - exit pairs explicitly; because there are hammocks with n vertices and $O(n)$ edges that have $O(n^2)$ subhammocks we cannot possibly expect to exhibit all possible entry - exit pairs within a linear number of steps.

So far in our discussion we have made a number of assumptions about entries and exits. Let us discuss the definitions that we propose to satisfy these conditions and remain close to the intuitive idea of entries

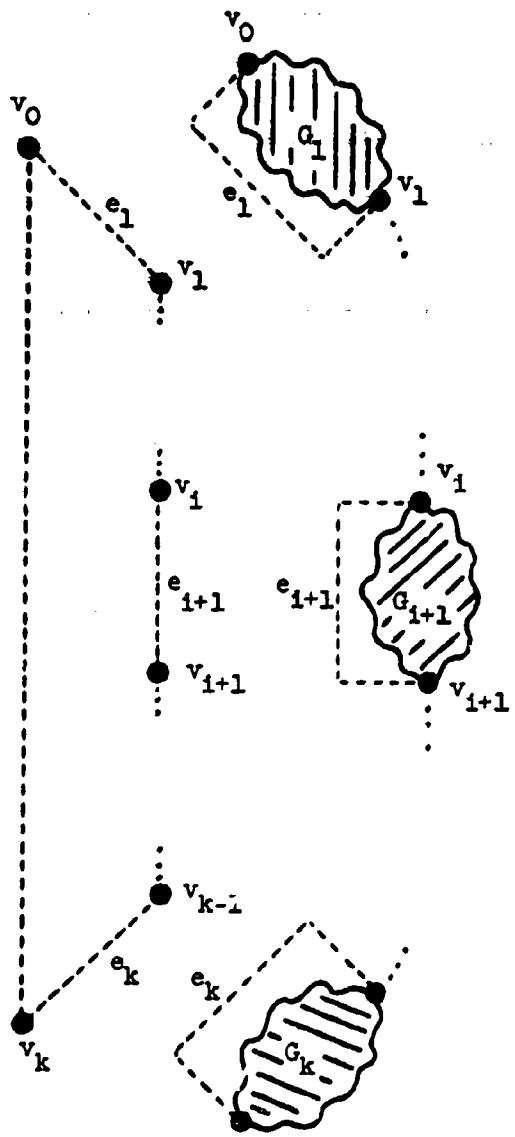


Figure 6.4. Entries and exits in a polygon.

and exits given in the previous section, before we present a complete example of our parsing algorithm.

Let G be a digraph, S be a subgraph of G and v a boundary vertex of S. We say that v is an entry (exit) of S if:

(i) every edge incident to v and to a vertex of $G-S$ enters (leaves) v; or if

(ii) every edge incident to v and to another vertex of G leaves (enters) v.

Figure 6.5 illustrates these definitions.

Ideally, one would like to use definitions of the following type: "a boundary vertex v of subgraph S is an entry (exit) if every path $\alpha \Rightarrow^* x$ $(x \Rightarrow^* \omega)$ in which $x \in S$ includes v." The problem with a definition of this kind is that in order to determine whether a vertex is an entry or an exit, one needs global information about the flow in the hammock -- precisely the information that our analysis is trying to uncover. We chose our definitions because they satisfy the following conditions:

(C1) A vertex that is an entry (exit) according to our definition will be an entry (exit) in the global sense given above as well. (The converse is not true, for instance, vertex 5 in the hammock of Figure 6.6.)

(C2) Replacement of a subhammock S of a hammock H by a single edge from the entry of S to its exit, transforms H into a hammock with the same start and finish vertex as H.

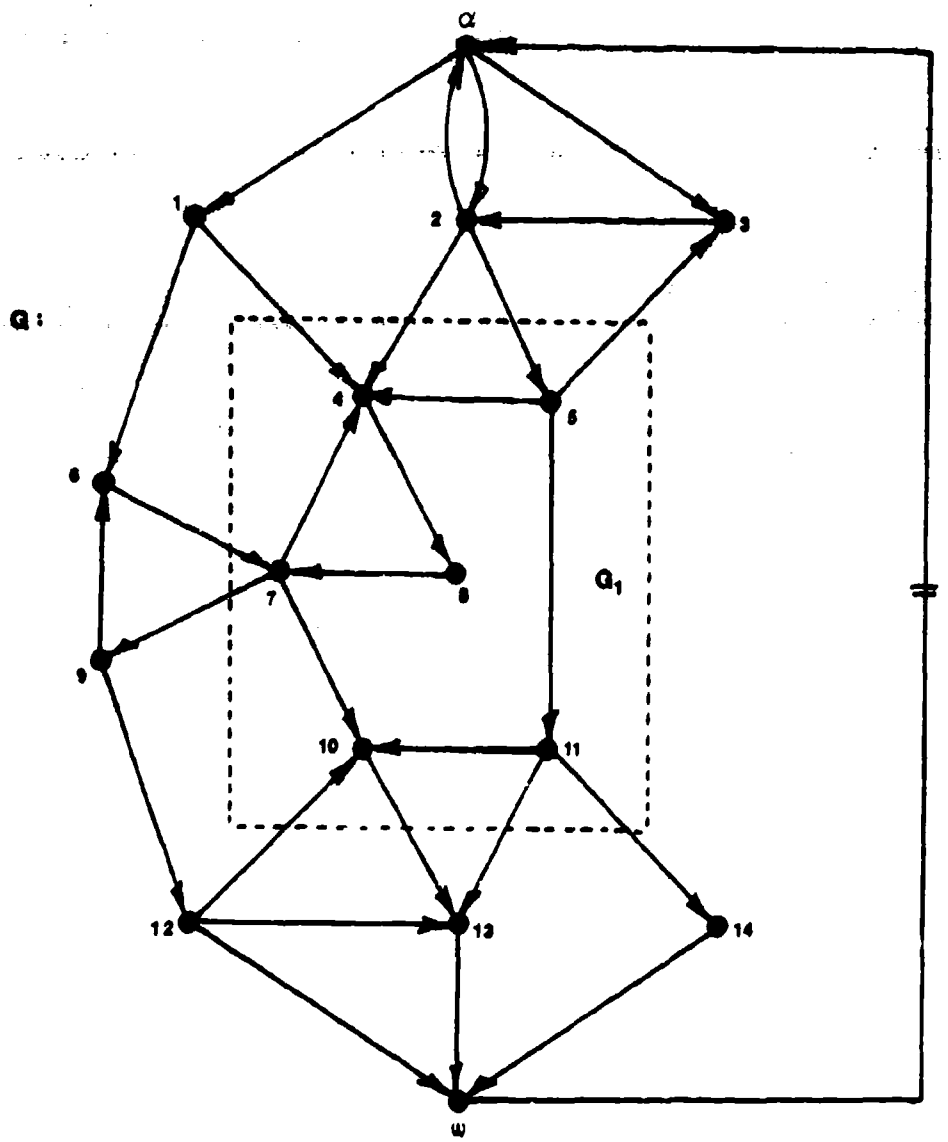


Figure 6.5. Vertices 4 and 5 are entries of G_1.

Vertices 10 and 11 are exits of G_1.

Vertex 7 is a boundary vertex of G_1 that is not an entry or an exit.

(C3) One can determine whether a boundary vertex v of a subgraph S is an entry or an exit by examining only edges incident to v knowing only about each edge whether it belongs to S or not.

(C4) If S_1 and S_2 are subgraphs such that $S_1 \cap S_2 = \{v\}$ and every edge incident to v belongs to S_1 or S_2, and if v is an exit (entry) of S_1 then it must be an entry (exit) of S_2.

(C5) If one reverses the directions of all the edges of a hammock its entries become exits and its exits entries.

In addition they include the definitions used by other authors as special cases. The conditions (C1), (C2), and (C3) are the assumptions made during the preceding discussion and are therefore "necessary" in some sense; the other conditions are merely nice features and the validity of our theory does not depend on them.

A complete example of our parsing algorithm can now be presented. Figure 6.6 shows a hammock H, $N(H)$, and the triconnected components of $N(H)$. Figure 6.7 shows the TCG T of $N(H)$, and two other tree structures derived from T, T_1 and T_p. The tree T_1 has been computed by adding to the label of each vertex of T a sequence of vertices as follows:

-- the sequence of a vertex of T_1 associated to a bond or triconnected graph contains the vertices of the separation pair of $N(H)$ that will be examined when we visit that vertex (in any order),

-- the sequence of a vertex of T_1 associated with a polygon contains the vertices of the polygon in the order in which we encounter them

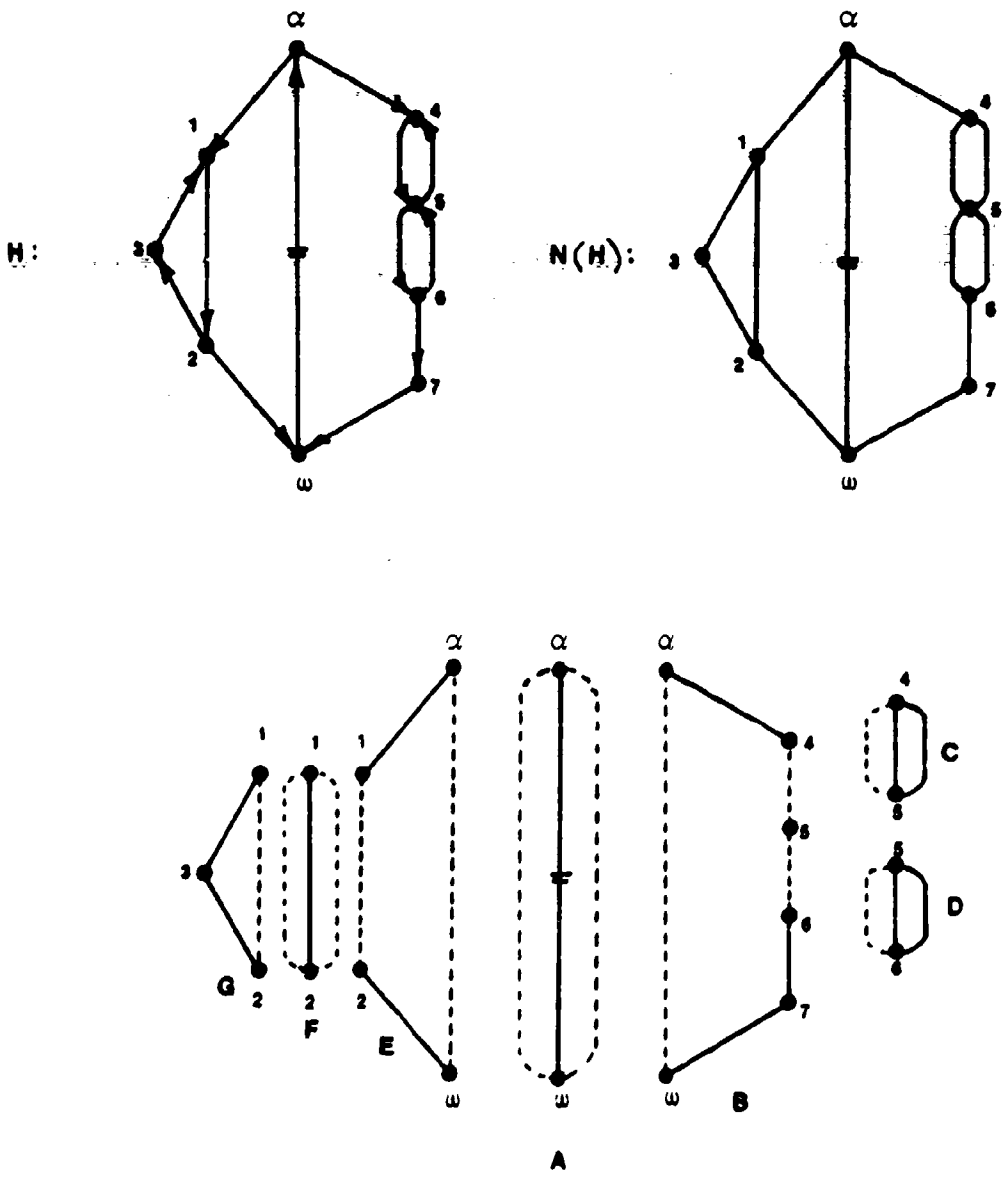


Figure 6.6. A hammock H , N(H) , and the triconnected components of N(H) .

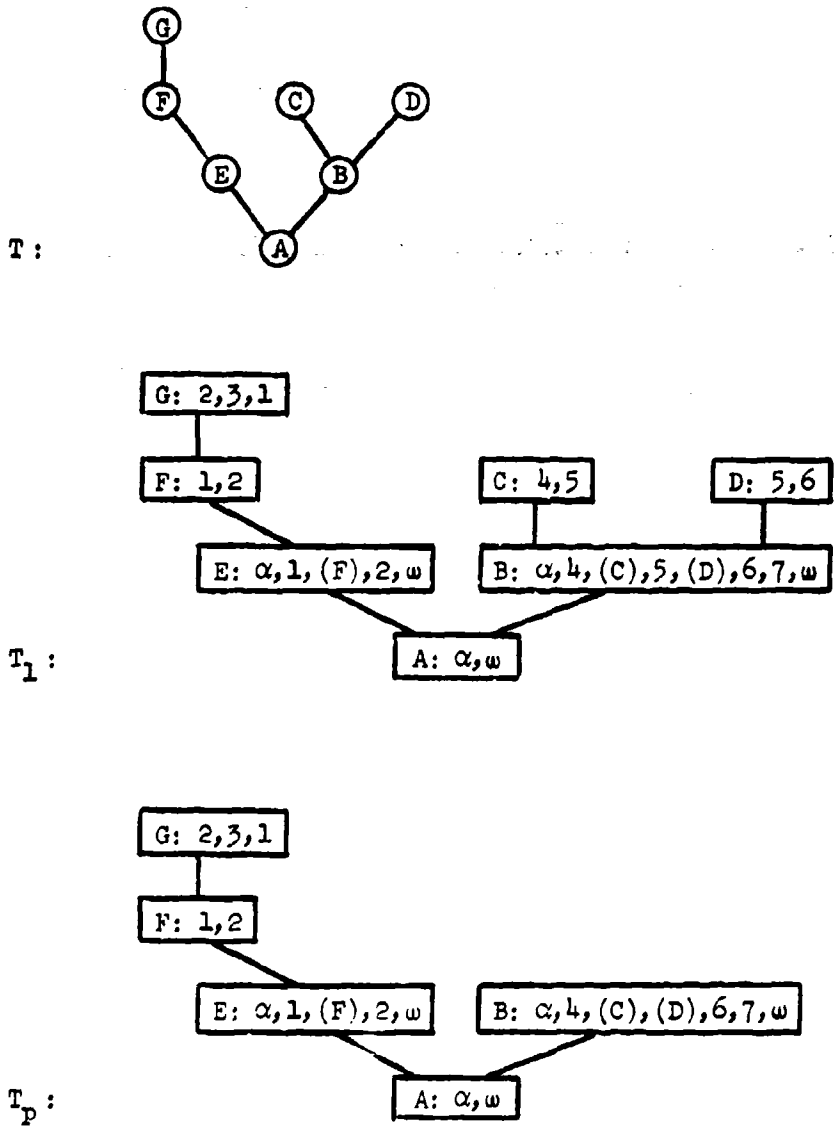


Figure 6.7. Structures used to obtain parses of hammocks.

as we go from one of the vertices of the separation pair corresponding to that polygon, to the other.

The tree T_p is computed from T_1 by

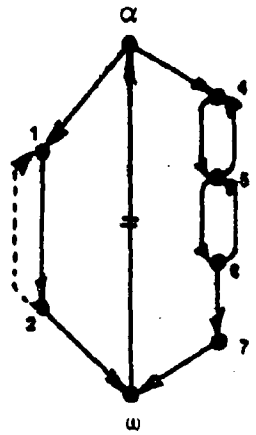
-- eliminating the separation pairs of $N(H)$ that are not entry - exit pairs of H. (For example, pairs $4,5$ and $5,6$);
-- eliminating the corresponding vertices from the sequences associated to polygons. (For example vertex 5 in the sequence of node B of T_1); and
-- ordering the sequences associated with the nodes so that for any entry - exit pair the entry appears before the exit.

Finally, Figure 6.8 shows how a parse of H can be read from T_p in a rather direct manner. The process traverses T_p from leaves to root "ridding" one or more replacements of prime subhammocks by single edges at each of the nodes it visits. These reductions are computed as follows (see figure for details):

-- if the node corresponds to a polygon
    -- eliminate the prime subhammocks lying between any two vertices of the sequence,
    -- eliminate the vertices of the polygon by what amounts to series reductions in H.
-- if the node corresponds to a bond or a triconnected graph, replace the prime subhammock determined by the two vertex sequence of the node by a single edge.

Given the property of Lemma 6.1, it is easy to see that the traversal from leaves to root guarantees that the hammocks being replaced in this

Processing of vertex G of T_p : - elimination of vertex 3 of H.



Processing of vertex B of T_p : - elimination of subhammock $C \cup D$ between 4 and 6;
- elimination of vertex 4 of H;
- elimination of vertex 6 of H;
- elimination of vertex 7 of H.

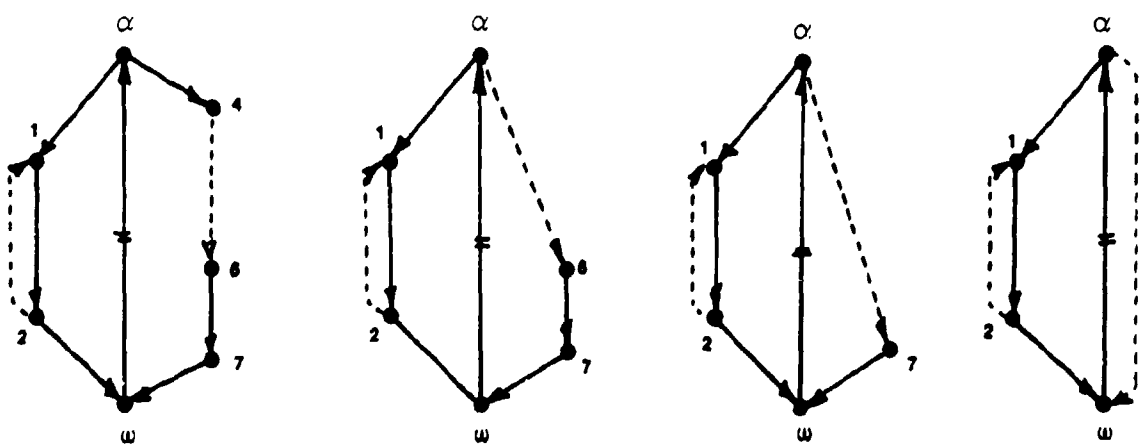
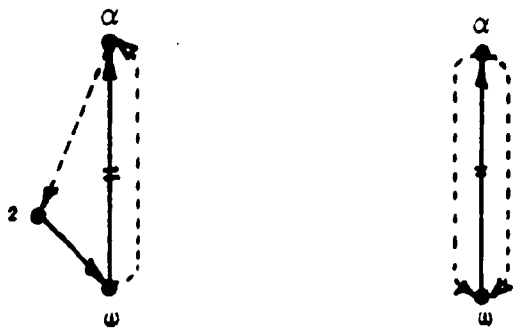


Figure 6.8. Example of the parse of a hammock.

Processing of vertex F of T_p : - elimination of subhammock between 1 and 2.

Processing of vertex E of T_p : - elimination of vertex 1 of H ;
- elimination of vertex 2 of H .



Processing of vertex A of T_p : - elimination of subhammock between α and ω.

Figure 6.8 (continued). Example of the parse of a hammock.

process are indeed prime. Thus, our discussion can be considered an informal proof of the following fact.

Theorem 6.1. The process just described can be implemented to produce a prime subhammock parse of a hammock H with n vertices and m edges in $O(n+m)$ steps provided that:

(i) $N(H)$ is biconnected; and

(ii) the definitions of entry and exit used satisfy (C2), (C3), and (C4). □

The only assumption that has not been discussed so far is the biconnectivity of $N(H)$. We will end this section by describing how the algorithm just described can be used in hammocks that do not satisfy this condition to provide similar structural information.

Figure 6.9 shows a hammock for which the assumption is not true. In such cases we proceed as follows:

-- We break $N(H)$ into biconnected pieces (see [AHO 76] for the description of an algorithm to perform this task in a number of steps proportional to the size of $N(H)$). Because α and ω are adjacent in $N(H)$, there will be a piece H_0 containing both.

-- The biconnected piece H_0 is then analyzed using the method described.

-- We remove H_0 from $N(H)$, obtaining in this way a set of connected multigraphs $G_1, G_2, \ldots, G_k$, each sharing an articulation point with H_0. Each of these multigraphs G_i is converted into a hammock H_i by "splitting" the articulation point, x_i, that it shares with H_0 into two vertices α_i, ω_i so that:

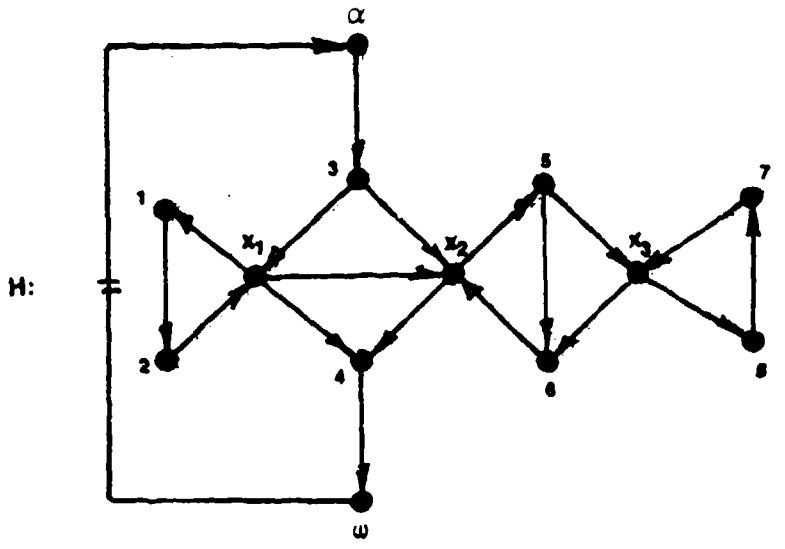


Figure 6.9. A hammock H for which N(H) would not be biconnected: x_1, x_2, and x_3 would be articulation points of N(H).

(i) every edge (y,x_i) of G_i becomes an edge (y,w_i) of H_i ; and

(ii) every edge (x_i,z) of G_i becomes an edge (d_i,z) of H_i .

The whole process is then applied recursively to the hammocks just computed. Figure 6.10 shows how this process would work on the hammock of Figure 6.9

Having accounted for this assumption, we have completed the description of our parsing algorithm for general hammocks. In the two sections that complete this chapter we will apply this algorithm to two subclasses of hammocks introduced by different authors to represent the flow of control of "reasonable" programs. We will discover that most of the problems that made our algorithm complicated in the general case disappear when one restricts in a natural way the set of hammocks that the algorithm is expected to handle.

6.3 Parsing Proper Programs.

A proper program is a hammock H in which every vertex v is of one of three types:

(i) a function node: in-degree(v) = out-degree(v) = 1 ; or

(ii) a predicate node: in-degree(v) = 1 and out-degree(v) = 2 ; or

(iii) a collect node: in-degree(v) = 2 and out-degree(v) = 1 .

An example of a proper program is shown in Figure 6.11. Note that if H is a hammock and every vertex of $N(H)$ has degree three or less, H must be a proper program.

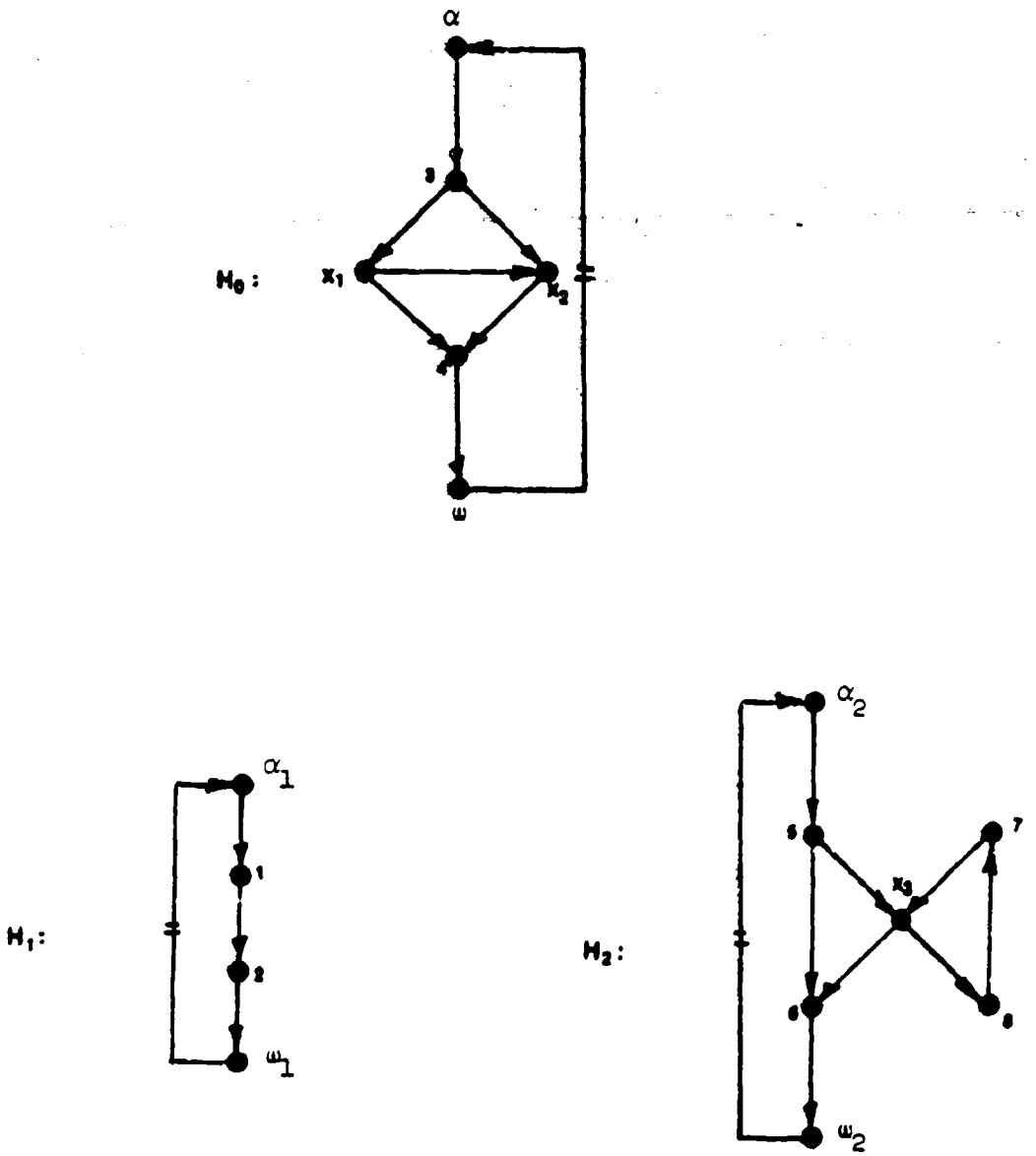


Figure 6.10. The hammock of Figure 6.9 is analyzed by analyzing H_0 using the algorithm described, then applying the process recursively to H_1 and H_2.

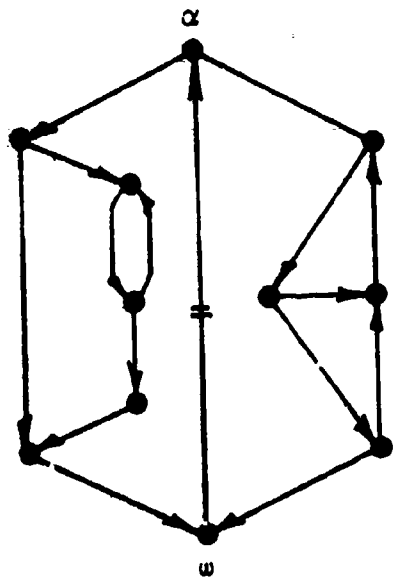


Figure 6.11. A proper program

According to Gannon and Hecht ([GAN 77]), this class of hammocks was introduced by Linger and Mills in conjunction with a method to analyze their flow of control that differs from all other standard approaches to the problem. In their paper, Gannon and Hecht present an algorithm that extracts -- in $O(n^3)$ steps for a proper program with n vertices -- the structural information needed to perform the flow analysis proposed by Linger and Mills. Fredrickson later refined the method of Gannon and Hecht into an algorithm that performs this task in $O(m\,\alpha(n,m))$ steps for a proper program with n vertices and m edges.

The purpose of this section is twofold. We want to show that many of the complications involved in the prime subhammock parse of general hammocks disappear when dealing with proper programs, and also exhibit the basic equivalence of our parsing method and that of Gannon and Hecht.

A property that makes proper programs particularly well suited to prime subhammock parsing is the following

Lemma 6.2. Let H be a proper program. $N(H)$ is biconnected.

Proof. [See Appendix C.] □

This lemma guarantees that when parsing proper programs we will not need to perform the repeated decomposition of the input into biconnected pieces described at the end of the previous section.

Steill greater simplifications can be achieved using the following facts:

Lemma 6.3. Let H be a proper program with start vertex α and finish vertex ω, and let S be a subhammock of H. The digraph H' obtained by replacing S by a single edge from its entry to its exit is a proper program with start vertex α and finish vertex ω.

Proof. [See Appendix C.] □

Lemma 6.4. Let H be a proper program and let S be a subgraph of $N(H)$ that does not include the return edge. The subgraph S can be reduced to a single edge by one series, parallel, or triconnected reduction (as defined in Chapter 3) if and only if the subgraph S' of H containing all the vertices and edges of S is a prime subhammock.

Proof. [See Appendix C.] □

These two lemmas imply that for every parse of $N(H)$ as a TT network (with return edge (α,ω)) using the Universal Replacement System, there is a prime subhammock parse of H. As a consequence, the prime subhammock parses of H can be read from the decomposition tree of $N(H)$ as described in Section 3.4, and we can avoid the process of testing the separation pairs of $N(H)$ as to whether they are entry - exit pairs of H.

An example of the simplified version of the parsing algorithm is shown in Figures 6.12 and 6.13. Figure 6.12 shows the process of obtaining a decomposition tree of $N(H)$ for a proper program H, and Figure 6.13 illustrates the process of reading a prime subhammock parse of H from the decomposition tree of $N(H)$.

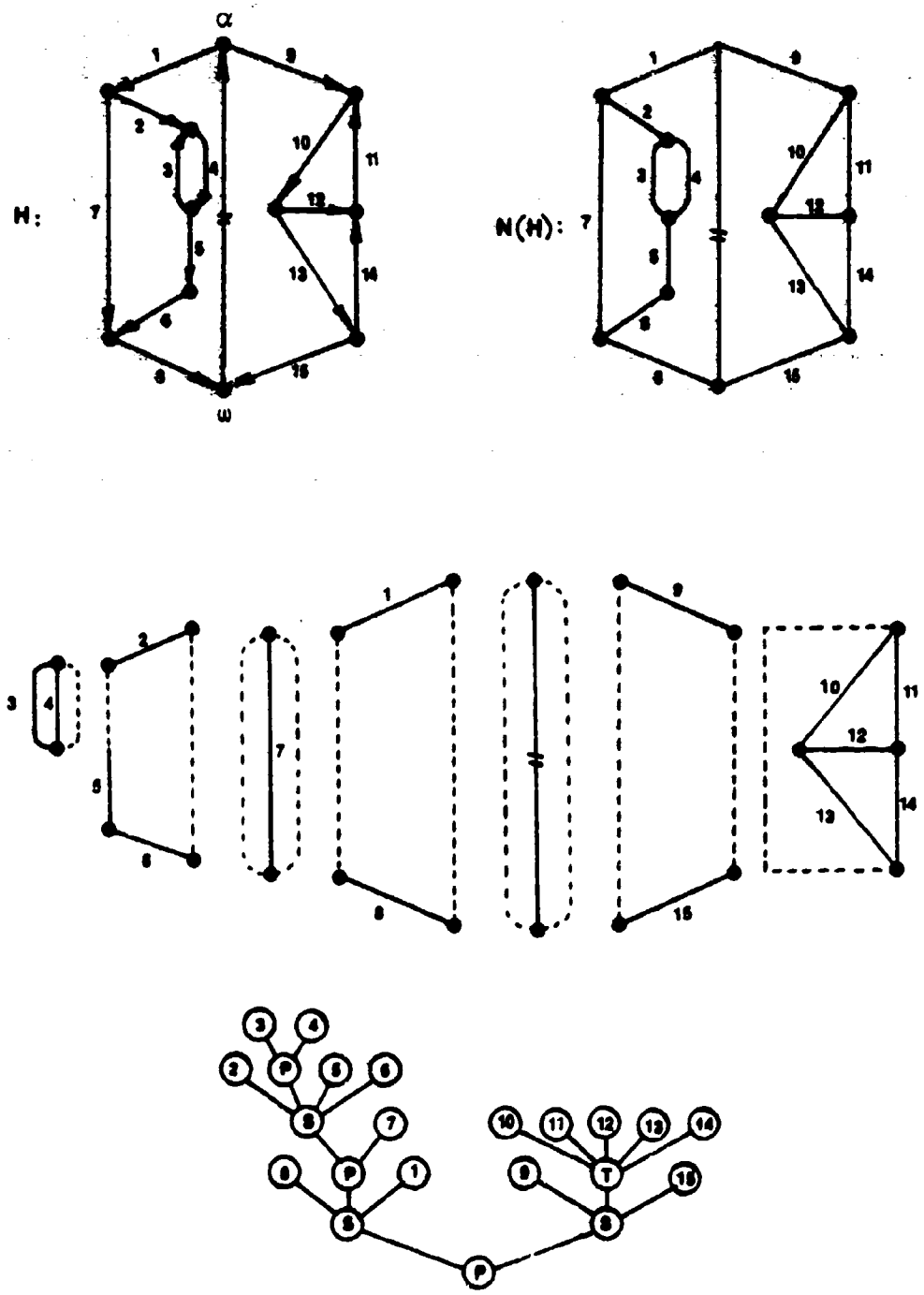


Figure 6.12. A proper program H , its undirected multigraph H_u and a decomposition tree of H_u obtained from its triconnected components.

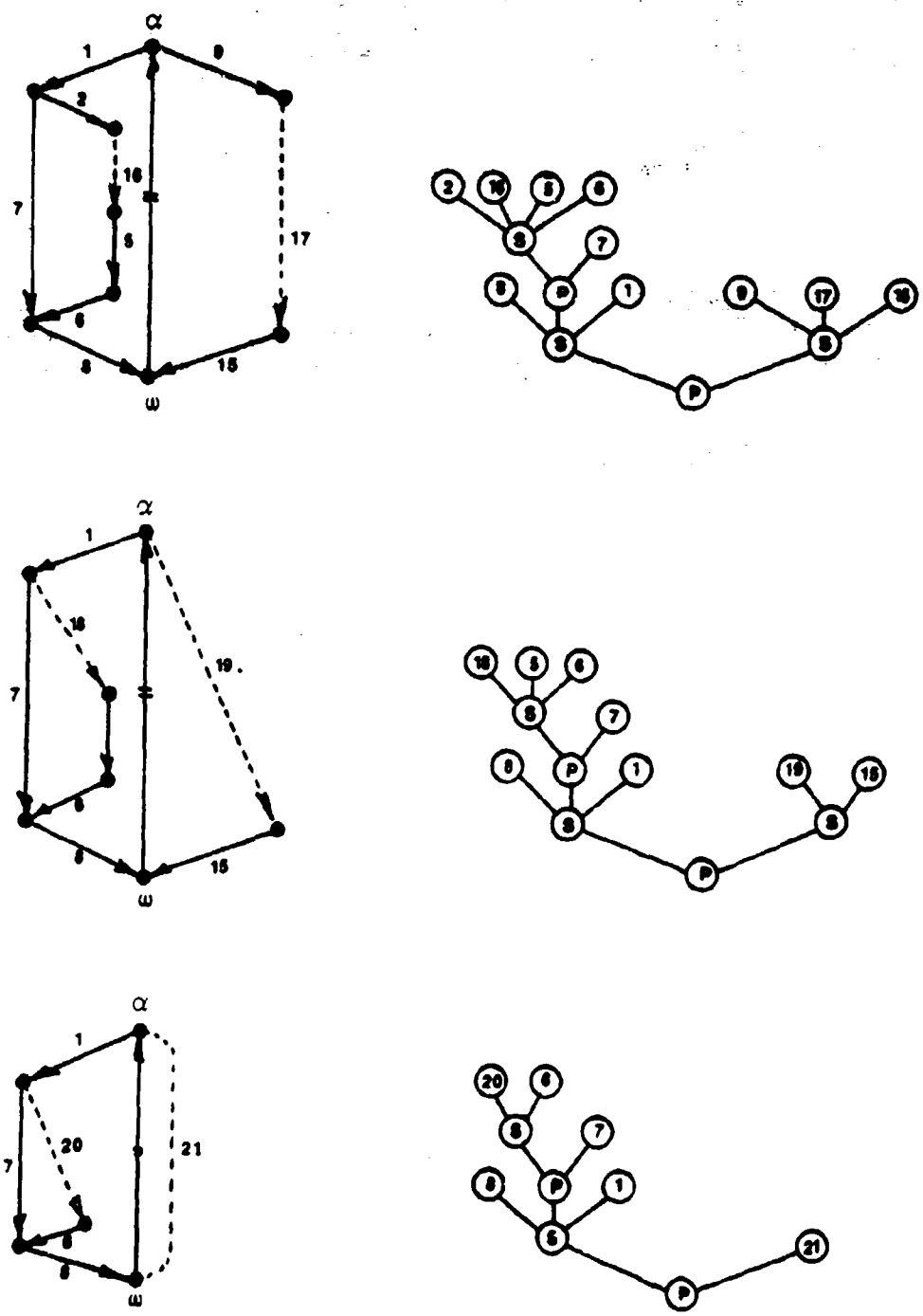


Figure 6.13. Parsing a proper program.

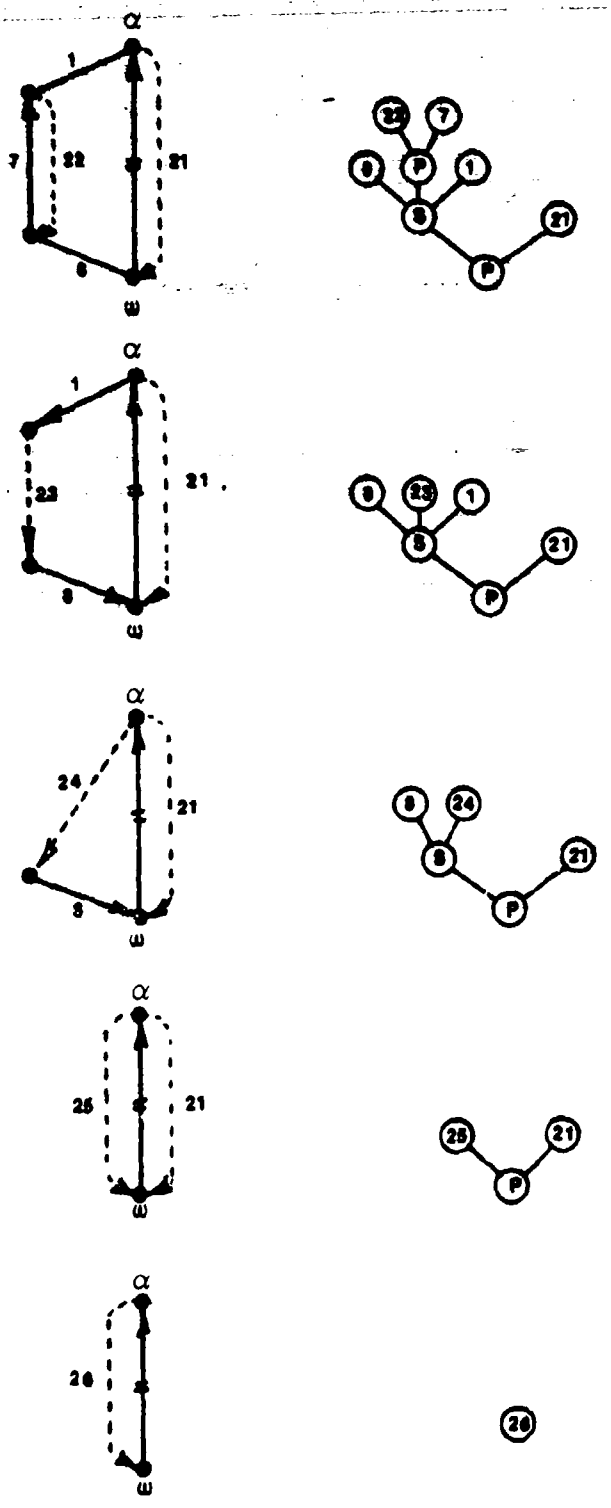


Figure 6.13 (continued). Parsing a proper program.

The relationship expressed by Lemma 6.4 can be combined with a result of Even and Tarjan ([EVE 76]) to show that our parsing approach is optimal in some sense. We will prove that any algorithm that performs the prime subhammock parse of proper programs can be used to compute the triconnected components of biconnected multigraphs in which the degree of each vertex is at most three. Our argument is the following.

Let G be a biconnected multigraph with n vertices and m edges, and let each vertex of G have at most degree three. Suppose that we can assign directions to the edges of G so it becomes a strongly connected digraph H. Because of the degree restriction on G, every vertex of H will be a function, predicate, or collect vertex, and H can be considered a proper program with any two adjacent vertices as start and finish. In addition $N(H) = G$ by construction, therefore -- according to Lemma 6.4 -- any prime subhammock parse of H gives us directly a parse of G using the Universal Replacement System from which one can trivially compute the decomposition tree of G. Let us then complete our argument by showing how the transformation of G into H can be achieved using the algorithm of Even and Tarjan in $O(n+m)$ steps.

Given a biconnected multigraph G with n vertices and m edges, and one of its edges (u,v), Even and Tarjan show how to number the vertices of G from 1 to n in $O(n+m)$ steps so that:

(a) v is assigned number one;

(b) u is assigned number n; and

(c) any vertex (except u and v) is adjacent both to a higher-numbered and to a lower-numbered vertex.

Let us now assign directions to the edges of G so they go from lower-numbered to higher-numbered vertices with the exception of (u,v) which goes from u to v. For any vertex x of the digraph H thus computed there must be a path $x \Rightarrow^* u$ in H because otherwise there would be a highest-numbered vertex z reachable from x and z would not satisfy condition (c). A similar argument shows that there must be a path $v \Rightarrow^* x$ as well and these two facts are enough to prove that H must be strongly connected since for any two of its vertices x, y, there will be paths $x \Rightarrow^* u$ and $v \Rightarrow^* y$ in H and thus (with the edge (u,v)) there will be a path $x \Rightarrow^* y$.

We can therefore conclude the following:

Theorem 6.2. The task of computing a prime subhammock parse of a proper program is equivalent to computing the triconnected components of a biconnected multigraph whose vertices have at most degree three. □

Let us conclude this section by considering the relationship between the parses of proper programs produced by our method and those produced by the algorithm of Gannon and Hecht.

Gannon and Hecht analyze proper programs in terms of prime subprograms, which are defined as non-trivial subhammocks that are either (i) maximal sequences of function nodes or (ii) do not properly contain any non-trivial subhammock. The prime subprogram parsing of a proper program is accomplished by identifying its prime subprograms, replacing each of them by a single function node, and repeating the process on the resulting proper program. Figure 6.14 shows the prime subprogram parsing of the proper program of Figure 6.12.

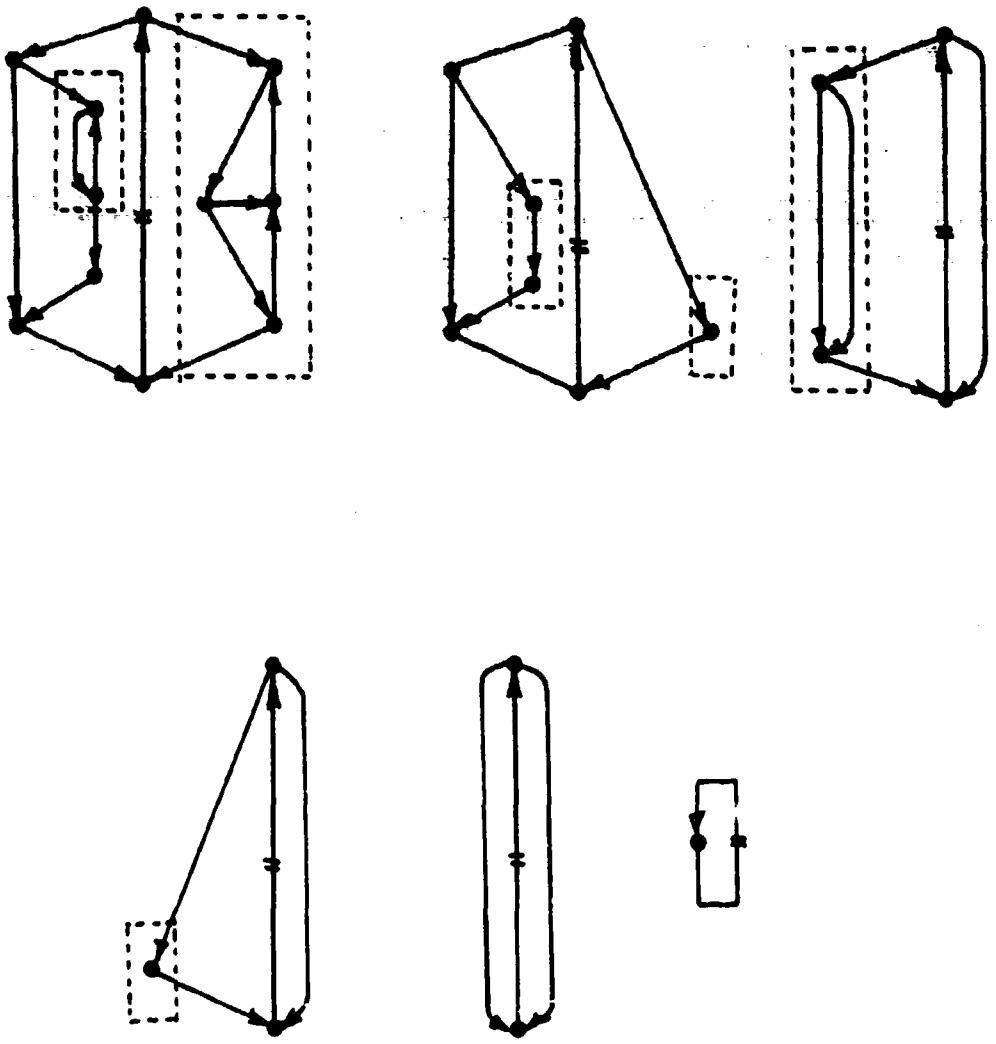


Figure 6.14. Parse of the proper program of Figure 6.12 by prime subprograms. At each stage the prime subprograms are identified by dotted boxes.

Let us consider prime subhammock parses of proper programs in which:

(i) following every replacement of a prime subhammock whose corresponding component of $N(H)$ is a bond or a triconnected graph we immediately eliminate either its entry or its exit by a series reduction, as shown in Figure 6.15; and in which

(ii) we postpone the elimination of any vertex of a polygon (except those eliminated in (i)) until every pair of consecutive vertices on the polygon are joined by a single edge, at which point we eliminate them all "at once" (so to speak) by series reductions.

From a prime subhammock parse that satisfies these restrictions, one can trivially obtain a proper program parse by grouping the prime subhammock reductions. Conversely by interpreting each prime subprogram reduction as several prime subhammock reductions, one can obtain a prime subhammock parse from any prime subprogram parse.

We can use this basic equivalence of the two parsing methods and the fact that a proper program with n vertices can have at most $O(n)$ edges (due to the degree reduction of its vertices) and conclude this section by stating the following theorem.

Theorem 6.3. The triconnected components algorithm can be used to obtain a prime subhammock parse (or a prime subprogram parse) of a proper program having n vertices in at most $O(n)$ steps. □

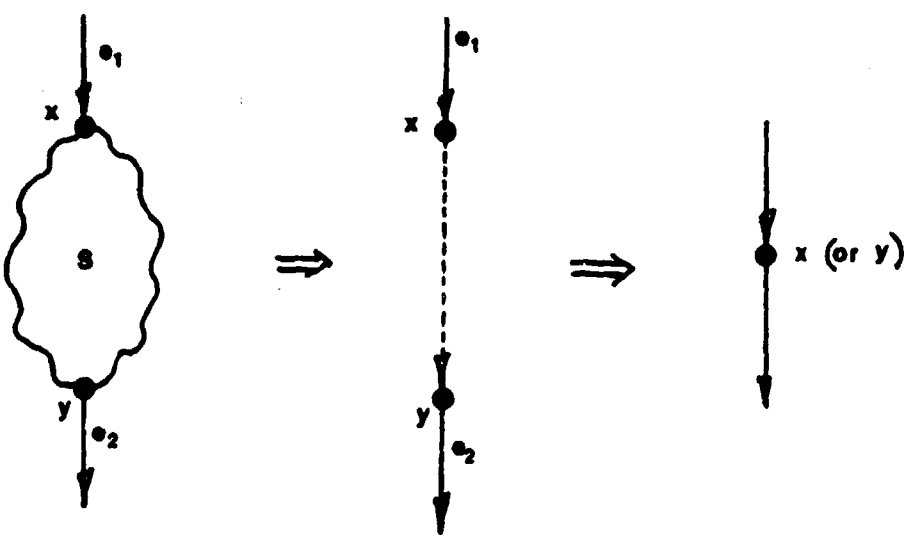


Figure 6.15. Interpretation of the replacement of a prime subprogram S that is either a bond or a triconnected graph by a single function code as a two step process of prime subhammock replacement.

6.4 Parsing Structured Programs.

Structured programming is the name given by Dijkstra ([DAH 72]) to a programming methodology he proposed aimed at establishing a close relationship between the tasks of writing a program and proving its correctness. Dijkstra's approach to this problem was to restrict the programmer to the use of a small set of simple constructs when writing a program.

Although the name "structured programming" has become fashionable and has been used to define many different programming techniques, we will employ it in its original sense. We will say that a hammock is a structured program if it can be generated from the pseudo-hammock of Figure 6.16(a) by repeated "expansion" of any vertex v such that in-degree(v) = out-degree(v) = 1 into one of the hammocks of Figure 6.16(b). (We have refrained from calling the graph of Figure 6.16(a) a hammock because it does not have distinct start and end vertices.) An example of the generation of a structured program is shown in Figure 6.17.

All the hammocks shown in Figure 6.16(b) with the exception of the one labelled case-of are proper programs, therefore all structured programs built with these constructs will be proper programs as well. It is therefore not too surprising that -- even if one uses the case-of construct -- structured programs share with proper programs the properties that allowed us to parse them using a simpler version of the general prime subhammock parsing algorithm. In particular:

Lemma 6.5. Let H be a structured program. $N(H)$ is biconnected.

Proof. [See Appendix C.] □

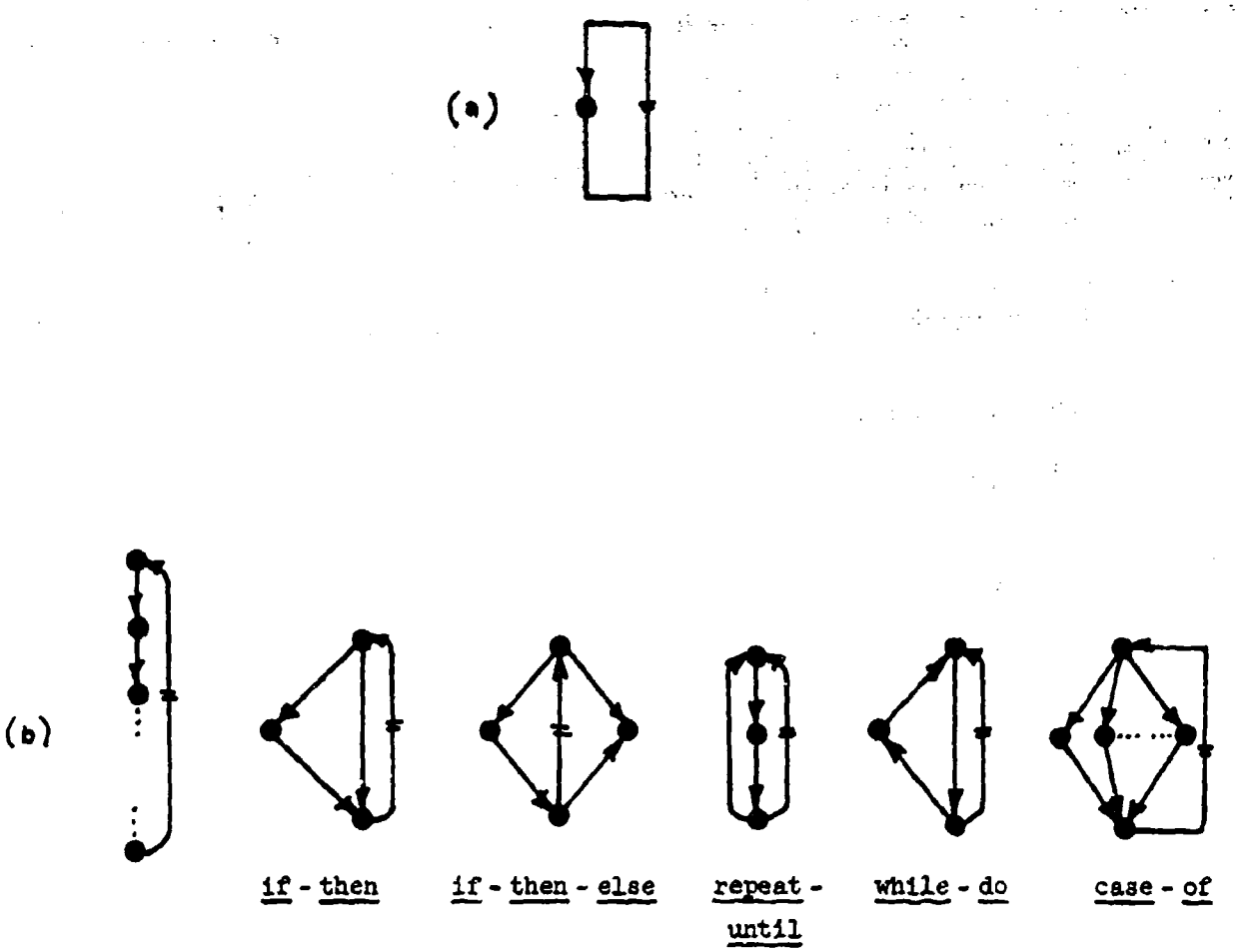


Figure 6.16. Definition of structured programs. The hammocks of part (b) are named by the program constructs from which they originate.

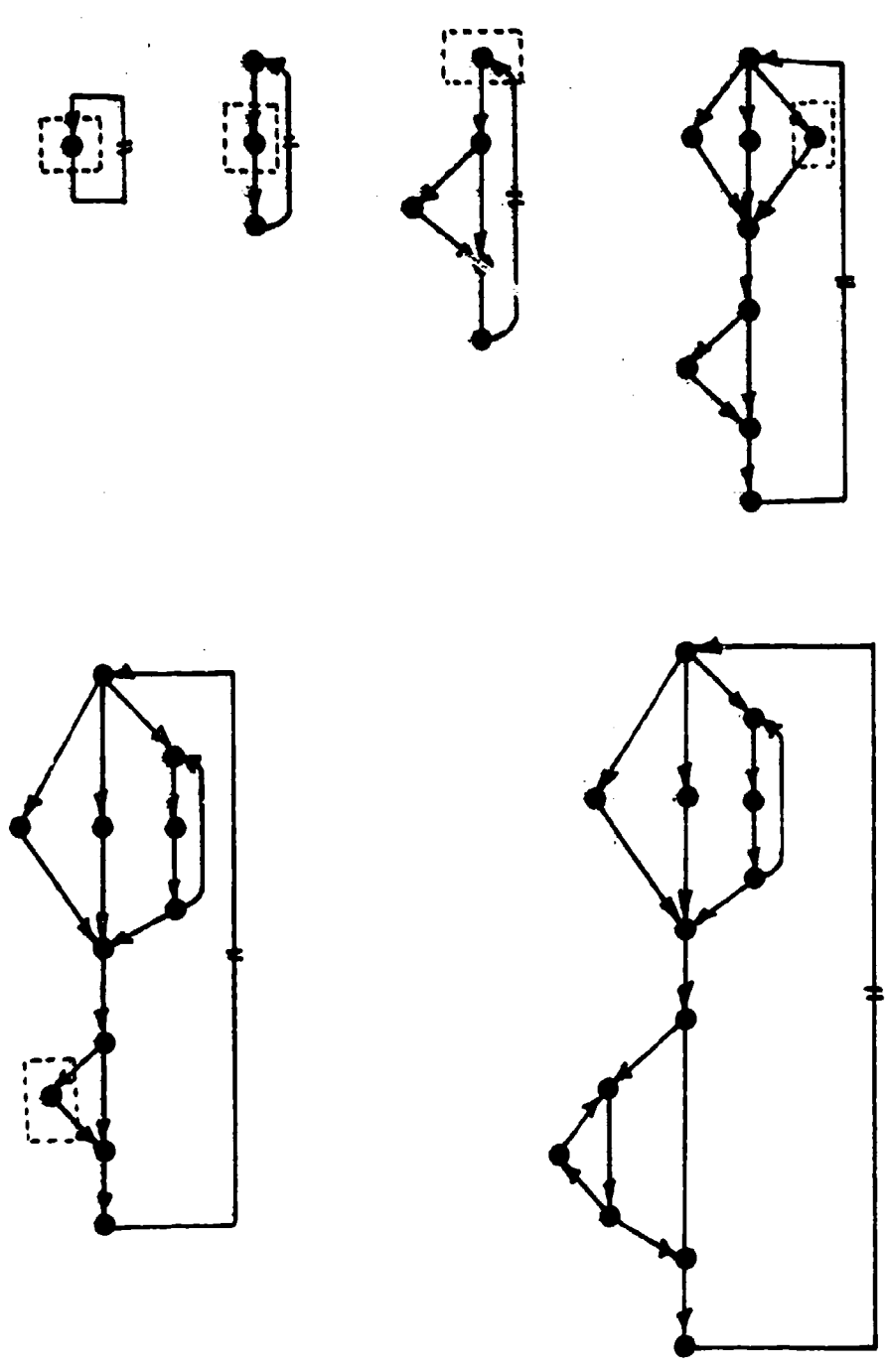


Figure 6.17. Construction of a structured program. At each step the vertex being replaced is marked by a dotted box.

However, structured programs allow us to simplify the prime subhammock parsing algorithm even further than proper programs did, mainly as a consequence of the following.

Lemma 6.6. Let H be a structured program. $N(H)$ can be reduced to a double bond by a sequence of series and parallel reductions that do not involve the return edge.

Proof. [See Appendix C.] □

Basically this lemma tells us that $N(H)$ is a TTSP network with the edge (α,ω) being its return edge. (Note however that this does not mean that H is a TTSP multidigraph!) The most important consequence of this fact for our purposes is that we do not need to use the triconnected components algorithm to obtain the decomposition tree of $N(H)$ if H is a structured program. Instead, Algorithm 4.2 (modified to work on undirected graphs) may be used. This replacement does not improve the asymptotic bound on the number of steps needed to parse a hammock, but allows the use of a much simpler program.

It should be noticed that programs whose control flow graph is a structured program are likely to be very well suited to Rosen's approach of obtaining structural information during their syntax analysis. Therefore, even though the prime subhammock parses of structured programs are very simple to obtain, as we have seen, there may well be better ways of analyzing their flow of control.

Chapter 7. Summary of Results and Open Problems.

In our opinion these are the main results contained in this thesis ranked (loosely) in order of importance:

1 -- An algorithm to recognize the class of GSP digraph in a linear number of steps (Chapter 5).

2 -- An approach to the structural analysis of flow graphs suitable to be implemented to run in a linear number of steps using the triconnected components algorithm (Chapter 6).

3 -- A unified view of the theory of TT networks and the classical results about TTSP networks from an algorithmic point of view by relating them to the theory of triconnected decomposition of biconnected graphs (Chapters 3 and 4).

During our presentation of these results, a number of problems were discussed briefly only from the point of view of our immediate goals. Several of these problems are interesting on their own and probably merit further study. The list that follows contains our suggestions for further work derived from our results:

1 -- The use of the unique decomposition trees of MSP, TSP, and TTSP digraphs to solve the subgraph isomorphism problem for these classes of digraphs. This problem seems to lead directly into several interesting generalizations of the subtree isomorphism problem that are -- as far as we know -- unsolved.

2 -- The generalization of the method employed to perform the transitive reduction of GSP digraphs to produce the transitive reduction of k-dimensional orders.

3 -- The design of an efficient algorithm that uses the structural information produced by the prime subhammock parse of a hammock to perform the global flow analysis on it.

4 -- The design of an algorithm that parses proper programs in linear time without using the general triconnected components algorithm. (In other words, find a simplified triconnected components algorithm that works for biconnected multigraphs whose vertices have degree at most three.)

References

[AHO 72] A. V. Aho, M. R. Garey, and J. D. Ullman, "The transitive reduction of a directed graph," SIAM J. Comput. 1, 2 (June 1972), 131-137.

[AHO 76] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Mass., 1976.

[ALL 70] F. E. Allen, "Control flow analysis," ACM SIGPLAN Notices (Newsletter) 5, 7 (July 1970), 1-19.

[COC 70] J. Cocke, "Global common subexpression elimination," ACM SIGPLAN Notices (Newsletter) 5, 7 (July 1970), 20-24.

[DAH 72] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Structured Programming, Academic Press, London, 1972.

[DUF 65] R. J. Duffin, "Topology of Series-Parallel networks," Journal of Math. Analysis and Applications 10, (1965), 303-318.

[EVE 76] S. Even and R. E. Tarjan, "Computing an st-numbering," Theoretical Computer Science 2, (1976), 334-344.

[FRE 78] G. N. Fredrickson, personal communication.

[GAN 77] J. D. Gannon and M. S. Hecht, "An $O(n^3)$ algorithm for parsing a proper program into its prime subprograms." (Manuscript.)

[GRA 76] S. L. Graham and M. Wegman, "A fast and usually linear algorithm for global flow analysis," Journal ACM 23, 1 (January 1976), 171-202.

[HAR 60] F. Harary and R. Norman, "Some properties of line digraphs," Rendiconti del Circolo Matematico Palermo 9, (1960), 149-163.

[HAR 71] F. Harary, Graph Theory, Addison-Wesley, Reading, Mass., 1971.

[HAR 72] F. Harary, J. Krarup, and A. Schwenk, "Graphs suppresible to an edge," Canadian Math. Bull. 15, 2 (1972), 201-204.

[HARR 65] M. A. Harrison, Introduction to Switching and Automata Theory, McGraw-Hill, New York, 1965.

[HEC 72] M. S. Hecht and J. D. Ullman, "Flow graph reducibility," SIAM J. Comput. 1, 2 (June 1972), 188-202.

[HEC 74] M. S. Hecht and J. D. Ullman, "Characterizations of reducible flow graphs," Journal ACM 21, 3 (July 1974), 367-375.

[HEC 77] M. S. Hecht, Flow Analysis of Computer Programs, Elsevier North-Holland, 1977.

[HEM 72] R. Hemminger, "Line digraphs," in Graph Theory and Applications, Y. Alavi, D. R. Lick, and A. T. White (eds.), Springer-Verlag, Berlin, 1972, 149-163.

[HOP 72] J. R. Hopcroft and J. D. Ullman, "An n log n algorithm for detecting reducible graphs," in Proc. Sixth Annual Princeton Conf. on Inf. Sciences and Systems, Princeton, N.J., 1972, 119-122.

[HOP 73] J. E. Hopcroft and R. E. Tarjan, "Dividing a graph into triconnected components," SIAM J. Comput. 2, 3 (September 1973), 135-158.

[KAS 75] V. N. Kas'janov, "Distinguishing hammocks in a directed graph," Soviet Math. Dokl. 16, 2 (1975), 448-450.

[KEN 71] K. Kennedy, "A global flow analysis algorithm," Intern. Journal of Computer Math., Section A, 3 (1971), 5-15.

[KLE 75] J. B. Klerlein, "Characterizing line dipseudographs," Proceedings of the Sixth Conference on Combinatorics, Graph Theory, and Computing, 429-442.

[KNU 69] D. E. Knuth, The Art of Computer Programming, vol. 1, Fundamental Algorithms, Addison-Wesley, Reading, Mass., 1969.

[LAW 60] E. L. Lawler and G. A. Salton, "The use of parenthesis-free notation for the automatic design of switching circuits," IRE Transactions on Electronic Computers, EC-9 (September 1960), 342-352.

[LAW 77] E. L. Lawler and B. D. Swazlian, "Minimization of time varying costs in single machine scheduling." (Manuscript.)

[LAW 78] E. L. Lawler, "Sequencing jobs to minimize total weighted completion time subject to precedence constraints," Annals of Discrete Math., (to appear).

[LAW 79] E. L. Lawler, R. E. Tarjan, and J. Valdes, "The analysis and isomorphism of General Series Parallel digraphs," (in preparation).

[LIU 77] P. C. Liu and R. C. Geldmacher, "A O(n) graph reducibility algorithm using depth-first search." (Manuscript.)

[MAT 78] D. W. Matula, "Subtree isomorphism $O(n^{5/2})$," Annals of Discrete Math. (to appear).

[MOM 77] C. L. Momma and J. B. Sidney, "A general algorithm for optimal job sequencing with Series-Parallel constraints," Tech. Report No. 347, School of Operations Research and Industrial Engineering, Cornell University, Ithaca, N.Y., July 1977.

[PRA 78] B. Prabhala and R. Sethi, "Efficient implementation of expressions with common subexpressions," presented at the POPL Conference, January 1978.

[RIO 42] J. Riordan and C. E. Shannon, "The number of two terminal Series Parallel networks," Journal of Math. Physics 21 (August 1942), 83-93.

[ROS 73] B. K. Rosen, "Tree manipulating systems and Church - Rosser theorems," Journal ACM 20, 1 (January 1973), 160-187.

[ROS 77] B. K. Rosen, "High level data flow analysis," Communications ACM 20, 10 (October 1977), 712-724.

[SCO 65] R. E. Scott, Elements of Linear Circuits, Addison-Wesley, Reading, Mass., 1965.

[SET 74] R. Sethi, "Testing for the Church - Rosser property," Journal ACM 21, 4 (October 1974), 671-679.

[SID 76] J. B. Sidney, "The two machine flow line problem with Series - Parallel precedence relations," Working paper 76-19, Faculty of Management Sciences, University of Ottawa, November 1976.

[SMI 78] B. J. Smith and H. F. Jordan, "Implications of Series - Parallel sequencing rules," Computing 19, 3 (1978), 189-201.

[SZY 77] T. G. Szymanski and J. D. Ullman, "Evaluating relational expressions with dense and sparse arguments," SIAM J. Comput. 6, 1 (March 1977), 109-122.

[TAR 72] R. E. Tarjan, "Depth-first search and linear graph algorithms," SIAM J. Comput. 1, 2 (June 1972), 146-160.

[TAR 74] R. E. Tarjan, "Testing flow reducibility," Journal of Comp. and Systems Sciences 9, 3 (December 1974), 52-53.

[TAR 75] R. E. Tarjan, "Solving path problems on directed graphs," Technical Report STAN-CS-75-528, Computer Science Department, Stanford University (1975).

[TAR 77] R. E. Tarjan, "Complexity of combinatorial algorithms," SIAM Review 20, 3 (July 1978), 457-491.

[TUT 66] W. T. Tutte, Connectivity in Graphs, University of Toronto Press, 1966.

[WAL 78] T. R. S. Walsh, "Counting labelled three-connected and homeomorphically irreducible two-connected graphs." (Manuscript.)

[WEI 71] L. Weinberg, "Linear graphs: theorems, algorithms, and applications," in Aspects of Network and System Theory, R. E. Kalman and N. DeClaris (eds.), Holt, Rinehart, and Winston, New York, 1971.

[WEI 75] L. Weinberg and P. Slepian, "Series-Parallel networks," IRE Transactions on Circuits CT-4, (September 1975), 290.

Appendix A. Graph Theoretical Definitions.

A graph $G = \langle V, E \rangle$ consists of a finite set of vertices V, and a finite set of edges E. Edges are pairs of distinct vertices; if the edges of a graph are unordered pairs the graph is undirected and if they are ordered the graph is directed. We will abbreviate directed graph as digraph. An edge -- directed or undirected -- will be denoted (u,v). Note that edges of the form (u,u) have been explicitly forbidden; these edges are normally called loops.

If the set of edges of a graph may be a multiset, that is, if we allow one edge to appear several times, the graph will be called a multigraph. We will abbreviate directed multigraph as multidigraph.

If $e = (u,v)$ is an edge, u and v are the endpoints of e, vertices u and v are adjacent, edge e is incident to u and v, and vertices u and v are incident to e. If e is a directed edge, e leaves u and enters v, u is a predecessor of v and v is a successor of u.

In an undirected multigraph the degree of vertex u is the number of edges incident to u. Two edges having the same endpoints are called parallel, and two edges sharing exactly one endpoint are consecutive. Two consecutive edges whose common vertex has degree two are said to be in series.

In a multidigraph, the out degree of a vertex u is the number of edges that leave u, and the in degree of u is the number of edges that enter u. A source is a vertex whose in degree is zero, and a sink is a vertex whose out degree is zero. Two edges that leave the same vertex and enter the same vertex are called parallel. Two edges of the form (u,v), (v,w) are consecutive. Two consecutive edges incident to vertex v are in series if the in degree and out degree of v are both one.

We have used the same names for different concepts defined in directed and undirected graphs. No confusion should result from this fact because it will be clear from the context which of the concepts is being used.

A path $u_1 \overset{*}{\Rightarrow} u_n$ in a multigraph (directed or undirected) is a sequence of vertices $u_1, u_2, \ldots, u_n$ such that for any $1 < i \leq n$, (u_{i-1}, u_i) is an edge of the multigraph. The path includes the edges (u_{i-1}, u_i) for $1 < i \leq n$. The vertices u_1 and u_n are the endpoints of the path. If $u_1 = u_n$ the path is called a cycle. A path in which all vertices are distinct is called a simple path, a cycle in which all vertices are distinct -- except the endpoints -- is a simple cycle.

A multigraph (directed or undirected) that contains no cycles is called acyclic.

The length of a path $u_1, u_2, \ldots, u_k$ is $k-1$. The distance between two vertices u, v is the length of the shortest path $u \overset{*}{\Rightarrow} v$; the distance is undefined if no such path exists.

An undirected multigraph is connected if for any two distinct vertices u, v there is a path $u \overset{*}{\Rightarrow} v$ in the multigraph. An edge e of a connected multigraph is a bridge if there exists a pair of vertices u, v such that every path $u \overset{*}{\Rightarrow} v$ includes e. A vertex x of a connected multigraph is an articulation point if there exists a pair of vertices u, v such that u, v and x are distinct and every path $u \overset{*}{\Rightarrow} v$ includes x.

A connected multigraph is biconnected if it has no articulation points, or equivalently, if for any three distinct vertices x, u, v there is a path $u \overset{*}{\Rightarrow} v$ in the multigraph that does not include x.

Two vertices x, y of a biconnected multigraph $G = \langle V, E \rangle$ are a separation pair if there is a partition of E into classes $E_1, E_2, \ldots, E_k$ with $k \geq 2$ such that:

(a) two edges in the same class belong to at least one path that does not include x or y except possibly as endpoints;

(b) any path containing edges in two distinct classes includes x or y;

(c) $E_1, E_2, \ldots, E_k$ can be merged into two disjoint sets E' and E'' each one containing at least two edges.

A biconnected multigraph is <u>triconnected</u> if it has no separation pairs.

The <u>undirected version</u> of a directed multigraph is the undirected multigraph obtained by changing every ordered pair (u,v) in the directed edge multiset into an unordered pair (u,v). A directed multigraph is <u>connected</u> if its undirected version is connected, and it is <u>strongly connected</u> if for any two of its vertices u, v, there are paths $u \overset{*}{\Rightarrow} v$ and $v \overset{*}{\Rightarrow} u$.

A directed acyclic graph is <u>transitive</u> if there is an edge (u,v) in the digraph between any two vertices joined by a path $u \overset{*}{\Rightarrow} v$. The <u>transitive closure</u> of an acyclic digraph $G = \langle V,E \rangle$ is the digraph $G_T = \langle V,E_T \rangle$ where E_T is the minimal subset of $V \times V$ that includes E and makes G_T transitive.

An edge (u,v) of an acyclic digraph is <u>redundant</u> if there is a path $u \overset{*}{\Rightarrow} v$ not including the edge. This concept is not well defined for multidigraphs or digraphs with cycles, but in acyclic digraphs redundant edges can be identified unambiguously ([AHO 72]). A directed acyclic graph with no redundant edges is <u>minimal</u>. The <u>transitive reduction</u> of an acyclic digraph is the digraph obtained by removing all the redundant edges.

Two graphs $G' = \langle V',E' \rangle$ and $G'' = \langle V'',E'' \rangle$ are <u>isomorphic</u> if there is a one-to-one correspondence between their vertex sets that preserves adjacency. That is, $f: V' \to V''$ and $f^{-1}: V'' \to V'$ are both functions and

$(u,v) \in E'$ if and only if $(f(u),f(v)) \in E''$. This concept can be extended naturally to multigraphs by requiring that if there are k edges (u,v) in G', there must be exactly k edges $(f(u),f(v))$ in G''.

A multigraph (directed or undirected) $G' = \langle V',E' \rangle$ is a subgraph of another multigraph $G = \langle V,E \rangle$ if $V' \subseteq V$ and $E' \subseteq E$. The vertices of G' adjacent to vertices that are not in G_1 are called boundary vertices of the subgraph.

A multigraph (directed or undirected) $G' = \langle V',E' \rangle$ is an embedded subgraph of another multigraph $G = \langle V,E \rangle$ if we can obtain G' from G by a sequence of the following operations:

(a) removal of an edge;

(b) replacement of two edges in series (u,v), (v,w) by a single edge (u,v).

For any subset S of the vertex set of a multigraph G, the induced subgraph of S is the maximal subgraph of G with vertex set S. The implicit subgraph of S is the induced subgraph of S in G_T, the transitive closure of G.

A tree is an undirected graph that is connected and acyclic. Vertices of a tree are also called nodes.

A rooted tree is a tree with a distinguished vertex called the root. In a rooted tree the vertices of degree one other than the root are called leaves and all other vertices internal nodes. In a rooted tree there is a unique simple path from the root to any other vertex v; the vertices of this path are the ancestors of v, and the ancestor of v adjacent to v is its parent. If u is an ancestor of v, v is a descendant of u; if v is the parent of u, u is a child of v.

A subtree is a connected subgraph of a tree. In a rooted tree, the subtree rooted at v is the minimal subtree that contains v and all its descendants. A rooted tree is ordered if there is a total order in the set of children of any node.

A binary tree is a rooted ordered tree in which every internal node has exactly two children called the left and right children. The left (right) subtree of an internal node v of a binary tree T is the subtree of T rooted at the left (right) child of v.

A subgraph G_1 of a multigraph G is a spanning tree of G, if G_1 is a tree and includes every vertex of G.

A graph $G = \langle V,E \rangle$ is complete if $E = V \times V$. The complete graph on n vertices is unique and it is normally denoted K_n.

A digraph $G = \langle V,E \rangle$ is complete bipartite if we can partition V into two subsets, H and T, such that (a) $H \cap T = \emptyset$, (b) $H \cup T = V$, and (c) $E = H \times T$. The set H is called the head and T the tail of G.

The line digraph ([HAR 60], [HEM 73], [KLE 75]) of a multidigraph $G = \langle V,E \rangle$ is the digraph $L(G) = \langle V_L, E_L \rangle$ constructed as follows:

- for each edge $e \in E$ there is a vertex $\ell(e) \in V_L$;
- for each pair of consecutive edges e_1, e_2 in G there is an edge $(\ell(e_1), \ell(e_2))$ in $L(G)$.

A multiple bond, or simply a bond, is a multigraph consisting of two vertices joined by one or more edges. A bond with two edges will be called a double bond, a bond with three edges a triple bond.

A polygon is a connected graph with k vertices and k edges, the edges forming a simple cycle including all the vertices. Polygons are named in the usual manner: triangle, quadrangle, pentagon, etc., according to the number of edges they contain.

Appendix B. The Efficiency of Algorithms.

We will study the efficiency of our algorithms by studying the number of steps that an abstract machine needs to execute them. This number of steps will be considered a function of some size measure of the input of the algorithm.

As an abstract model of the actual machines on which the algorithms will run we have chosen the Random Access Machine (RAM) (a detailed description of this machine can be found in Chapter 1 of [AHO 76]). We choose the RAM because it is a better model of most present day computers than most of the other abstract machines used in computation theory. In a RAM any logical, arithmetical or control operation takes one step but the numbers it manipulates are restricted in absolute value to some constant times the size of the input of the algorithm. This restriction on the size of the numbers manipulated corresponds to the fixed word size of most digital computers.

The inputs to all our algorithms are graphs or multigraphs. The size of a graph or multigraph will be normally measured by the number of elements in its vertex and edge set.

To simplify the derivation of the number of steps taken by a RAM running an algorithm we will discard constant factors by using the standard "big O" notation. We say that a function $f(x)$ is $O(g(x))$ if for some constants k_1 and k_2, $|f(x)| \leq k_1|g(x)| + k_2$ for all values of x. As an example of the use of this notation, if S is the size of the input of an algorithm, the absolute value of the numbers that the RAM may manipulate in a single step when running the algorithm can be expressed as $O(S)$.

Appendix C. Proofs of Lemmas and Theorems.

> proof, n. Evidence having a shade more of plausibility than of unlikelihood. The testimony of two credible witnesses as opposed to that of only one.
>
> - A. Bierce (The Devil's Dictionary)

This appendix provides the proofs of most of the results of the main body of the thesis. In order to avoid repetition we have proved first a few lemmas that are then referenced in later proofs.

Lemma C.1. A triconnected graph G with at least four vertices contains K_4 as an embedded subgraph.

Proof. We start by proving that there is a simple cycle of G that contains four or more vertices.

Select two adjacent vertices u, v of G. There has to be a path $u \Rightarrow^* v$ in G that does not contain (u,v) or that edge would be a bridge. If this path contains three or more edges we have found the desired cycle. Otherwise we have found a triangle x, u, v in G. In this case we proceed by selecting another vertex y of G: there must exist two paths a: $y \Rightarrow^* u$ not including x or v and b: $y \Rightarrow^* v$ not including x or v in G since it is a triconnected graph. Let w be the last vertex in a that is also in b (y is a candidate): there are disjoint paths $w \Rightarrow^* u$ and $w \Rightarrow^* v$ so we can repeat the argument given for y for vertex w and we conclude that G has a simple cycle with four vertices.

Therefore let C be a simple cycle of G that contains four or more vertices. We find two pairs of vertices x_1, x_2 and y_1, y_2 that split C into four pieces C_1, C_2, C_3, and C_4 as shown in Figure C.1(a). Because G is triconnected, there must exist a path $k: x_1 \Rightarrow^* x_2$ in G that does not include y_1 or y_2. Let x_1' be the last vertex in k that lies on C_1 or C_4 and x_2' the first vertex on k that lies in C_2 or C_3. Clearly the section of k between x_1' and x_2' does not contain any vertex of C except x_1' or x_2'. Similarly, there are vertices y_1' and y_2' lying in C and a path $y_1' \Rightarrow^* y_2'$. that does not include any vertex of C except y_1' and y_2' and such that y_1' is in C_1 or C_2 and y_2' is in C_3 or C_4. We therefore have the situation of Figure C.1(b) in which C_1', C_2', C_3', and C_4' form the cycle C and the paths C_5' and C_6' do not contain any vertices of C (except obviously x_1', x_2', y_1', and y_2').

If the paths C_5' and C_6' are disjoint, then we have clearly found an embedded K_4 in G. Otherwise let z be the vertex of C_6' closest to y_1' that belongs also to C_5'; we have the situation of Figure C.1(c) in which C_7' is the section of C_5' between x_1' and z and C_8' is the union of C_2' and C_3' and once again we have found an embedded K_4 in G. □

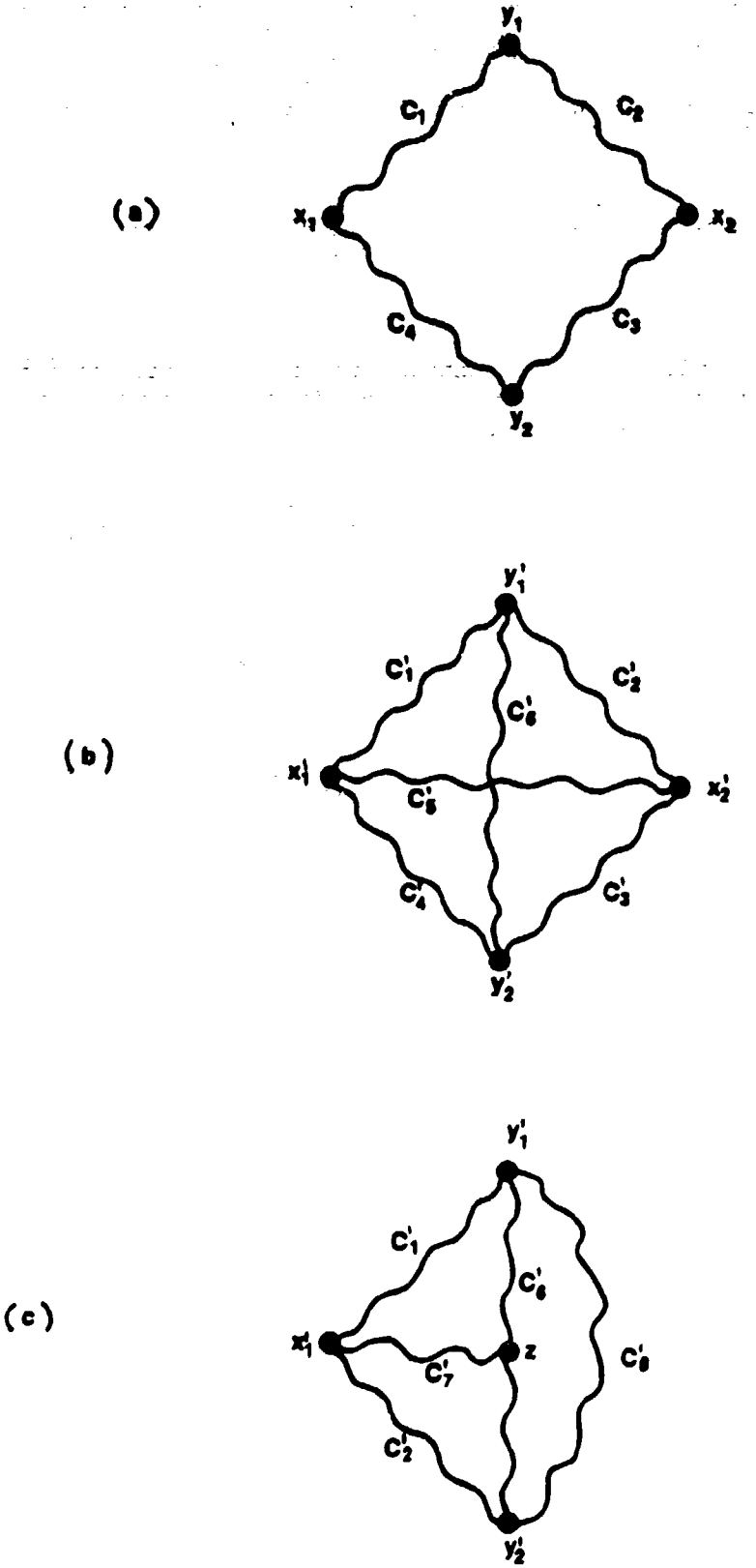

(a)
y_1
C_1
C_2
x_1
x_2
C_4
C_3
y_2
(b)
y_1'
C_1'
C_2'
C_6'
x_1'
x_2'
C_5'
C_4'
C_3'
y_2'
(c)
y_1'
C_1'
C_6'
x_1'
z
C_7'
C_8'
C_2'
y_2'

Figure C.1

Lemma C.2. Let G be a biconnected multigraph and S a triconnected component of G. S is an embedded subgraph of G.

Proof. Merge the triconnected components of G as much as possible without merging S with any other component. The result is a set of multigraphs $\{G_1, G_2, G_3, \ldots, G_k, S\}$ where each of the multigraphs G_i contains only one virtual edge (x_i, y_i) that it shares with S. All the multigraphs G_i are biconnected since they are obtained by merging some triconnected components of G (which are biconnected) and merging obviously preserves biconnectivity. Therefore there is a path $x_i \Rightarrow^* y_i$ in G_i that does not include the virtual edge (x_i, y_i) (otherwise (x_i, y_i) would be a bridge in G_i) and this path contains no virtual edges. Thus for any virtual edge (a,b) of S there is a path $a \Rightarrow^* b$ in G and all such paths are disjoint since each is contained in a different G_i and every actual edge of G appears in just one triconnected component. We therefore conclude that S can be obtained from G by removal of all the edges not on such paths followed by contraction of each of these paths to a single edge by series reductions.

Lemma C.3. A biconnected multigraph having a triconnected component that is not a bond or a polygon contains K_4 as an embedded subgraph.

Proof. Let G be the biconnected multigraph and G' one of its triconnected components that is a triconnected graph with at least four vertices. According to Lemma C.2, G' can be obtained from G by removal of edges and series reductions, and according to Lemma C.1, K_4 can be obtained from G' by a sequence of the same operations so we conclude that K_4 is an embedded subgraph of G. □

[Note: The transitivity of the embedded subgraph relation implied by the proof above is also used in a few other places without being formally proved.]

Lemma 3.2. Every subnetwork of a firmly connected TT network is firmly connected.

Proof. Let N_1 be a subnetwork of a firmly connected TT network N and let t_1 and t_2 be the terminals of N_1.

If N_1 does not contain any internal vertex, the proposition is trivially true. Otherwise let v be an internal vertex of N_1. Because v will also be an internal vertex of N, there is a terminal path p in N that includes v. Since p includes at least one internal vertex of N_1 and is a simple path it must include the only two boundary vertices t_1 and t_2 of N_1: one being the first vertex of N_1 in the path and the other being the last. The section of p between t_1 and t_2 is a terminal path of N_1 that includes v.

Therefore for every internal vertex of N_1 there is a terminal path in N_1 that includes v and N_1 is firmly connected. □

Lemma 3.3. A nontrivial indecomposible TT network is either a triangle, a triple bond, or a triconnected graph with at least four vertices.

Proof. Let $N = \langle V, E \rangle$ be a non-trivial, firmly connected, undecomposible TT network. Assume that N has a separation pair x, y, that is, we can find two subgraphs of N, $N_0' = \langle V_0, E_0 \rangle$ and $N_1' = \langle V_1, E_1 \rangle$ such that $V_1 \cup V_0 = V$, $V_1 \cap V_0 = \{x,y\}$, $E_0 \cup E_1 = E$, $E_0 \cap E_1 = \emptyset$, $\|E_1\| > 1$, and $\|E_2\| > 1$. Let N_0' contain the return edge of N. The TT networks

$N_0 = \langle V_0, E_0 \cup \{(x,y)\}\rangle$ with the return edge of N as its return edge, and $N_1 = \langle V_1, E_1 \cup \{(x,y)\}\rangle$ with (x,y) as its return edge, form a decomposition of N. This contradicts our assumption that N is undecomposible so we have to conclude that N is triconnected. Because N is also non-trivial, it must contain at least three edges. The only multigraphs that satisfy these conditions are triple bonds, triangles, and triconnected graphs with at least four vertices.

Lemma 3.4. Let N be a firmly connected TT network and N_1 a non-trivial proper subnetwork of N with boundary vertices x and y.

(a) x, y is a separation pair of N.

(b) There is at least one decomposition of N in which N_1 is a component.

Proof.

(a) The vertices x, y are a separation pair of N because we can split its edge set into the edges that belong to N_1 and those which do not. Each of these sets will contain at least two edges since N_1 is non trivial and is a proper subnetwork of N and any path including edges in both sets must contain x or y since they are the only boundary vertices of N_1.

(b) The proof of Lemma 3.3 given earlier proves this as well. □

Lemma 3.5.

(a) The TCG of a biconnected multigraph is a tree.

(b) No vertex of a TCG corresponding to a bond (polygon) can be adjacent to another vertex representing a bond (polygon).

Proof.

(a) Let G be the biconnected multigraph.

For any set of split components $G_1, G_2, \ldots, G_k$ of G, we define the Split Component Graph (SCG) associated with the set as having:

(i) a vertex for each split component,

(ii) an edge joining two vertices if and only if the split components corresponding to the vertices share a pair of edges generated in the same split operation.

Clearly the TCG of any biconnected multigraph G can be obtained from any SCG of G by "shrinking" all the edges of the SCG that join vertices associated with two triangles or two triple bonds. Because shrinking edges does not create or eliminate cycles or isolated vertices, and the TCG is unique, the TCG of G will be a tree if any SCG of G is a tree.

We will now prove that any SCG of a biconnected multigraph G is a tree by induction on the number of edges of G.

If G has only three edges it cannot be split, so its SCG is a trivial tree consisting of a single vertex and no edges. Assume that the proposition is now true for any biconnected multigraph with fewer than m edges, and let G have m edges. For any SCG, S, of G consider the first split operation performed to generate it; this operation produced two split graphs of G, G_1 and G_2, each one with fewer than m edges. Clearly S can be constructed from some SCGs, S_1 of G_1 and S_2 of G_2, by joining them by a single edge. The graphs S_1 and S_2 are trees by induction hypothesis, so S will also be a tree since it will be connected and any simple cycle of G has to be a cycle of either S_1 or S_2.

(b) If two such adjacent polygons or bonds can be found, an additional merge operation would produce a new set of triconnected components contradicting Theorem 3.2. □

Lemma 3.6. A TT network is Series Parallel if and only if it can be reduced to a double bond by an appropriate sequence of series and parallel reductions.

Proof. We prove first that if a TT network N is Series Parallel, it is reducible to a double bond by induction on the number of triconnected components of N .

If N has a single triconnected component, N is either a polygon or a bond and the proposition is trivially true. Let us assume that the proposition is true for any TT network with fewer than k triconnected components and let N have exactly k components. Let us consider the TCG of N as a rooted tree with the root being the vertex that corresponds to the triconnected component of N that includes the return edge, and select any leaf of this tree. The triconnected component of N associated with that vertex is a polygon or a bond (since N is TTSP) and contains exactly one virtual edge (since it is a leaf of the TCG of N). We can thus reduce this component to a double bond consisting of its virtual edge plus another edge arising from the reduction of all the other edges. The reductions identified in this manner, if applied to N , result in a TT network N' that can be obtained by merging all the triconnected components of N except the one associated with ℓ . Thus N' has k-1 components and is TTSP (since all its triconnected components are also triconnected components of ') and thus can be reduced to a double bond.

Since N' was obtained from N by series and parallel reductions, we conclude that N can be reduced as well.

We now prove that if N is not a TTSP network, it cannot be reduced to a double bond by series and parallel reductions.

If N is not TTSP, at least one of its triconnected components is a triconnected graph with four or more vertices. Thus, according to Lemma C.3, N contains K_4 as an embedded subgraph. This means that there are four vertices of N, x_1, x_2, x_3, x_4, such that any two of them are connected by a path $x_i \Rightarrow^* x_j$ $(1 \leq i < j \leq 4)$ and all six such paths are disjoint. Because three of the paths are incident to each vertex, and the paths are disjoint, none of these vertices can be immediately deleted by a series or parallel reduction. Furthermore, series and parallel reductions do not destroy paths between vertices that are not removed. This implies that any multigraph obtained from N by series or parallel reductions will contain K_4 as an embedded subgraph and therefore that N cannot be reduced to a double bond. □

Lemma 3.7. A firmly connected TT network is Series Parallel if and only if:

(i) it is a double bond, a triple bond, or a triangle,

or (ii) it has a decomposition whose core is a triple bond or a triangle and in which all the components (there are at most two) are TTSP networks.

Proof. We show first that if a TT network N satisfies (i) or (ii), it is a TTSP network.

If N satisfies (i) it is obviously TTSP since the set of its triconnected components contains only N itself. Otherwise let N_0,

N_1 , N_2 be the decomposition of N postulated by (ii) (there may be only one component but that fact is irrelevant for our argument). Because of the relationship between the operations of decomposing a TT network and splitting a biconnected multigraph given by Lemma 3.4, a set of split components of N can be obtained by the union of any three sets of split components of N_0 , N_1 , and N_2 , respectively. Any set of split components of N_1 or N_2 contains only triangles and triple bonds since both networks are TTSP, and the set of split components of N_0 contains only N_0 itself (a triangle or a triple bond). Consequently we have found a set of split components of N which contains only triangles and triple bonds. This implies that the triconnected components of N are only polygons and bonds and thus that N is TTSP.

We prove now the implication in the other direction: if a TT network N is TTSP then it satisfies (i) or (ii).

If N has fewer than four edges and is TTSP it must satisfy (i).

If N has more than three edges, consider any set of split components of N : because N is TTSP, all the members of the set are triangles or triple bonds.

Consider now the decomposition of N whose core is the split component that includes the return edge and whose components are the TT networks obtained by merging all the other split components among themselves as much as possible and selecting as return edge the virtual edge that the multigtaphs thus ootained share with the core. (Once again this decomposition may have a single component but the argument does not depend on this fact.)

This decomposition of N has a core that is a triangle or a triple bond and components that are TTSP networks since they have a split component set with only triangles and triple bonds, and therefore N satisfies (ii). □

Lemma 3.8. A firmly connected TT network is Series Parallel if and only if it does not contain K_4 (the complete graph on four vertices) as an embedded subgraph.

Proof. We prove first that if N is a TTSP network it does not contain K_4 as an embedded subgraph. We do this by showing that the triconnected components of N do not contain K_4 as an embedded subgraph and that the operation of merging biconnected multigraphs preserves this property.

If N is TTSP, all its components are polygons or bonds, therefore no component of N contains K_4 as an embedded subgraph.

Assume now that $G = \langle V, E\rangle$ is the result of merging the biconnected multigraphs $G_1 = \langle V_1, E_1\rangle$ and $G_2 = \langle V_2, E_2\rangle$ that share the virtual edge (x,y), and that neither G_1 nor G_2 contains K_4 as an embedded subgraph. By definition of merging: $V = V_1 \cup V_2$, $\{x,y\} = V_1 \cap V_2$, and $E = (E_1 - \{(x,y)\}) \cup (E_2 - \{(x,y)\})$. Thus there is a path $u \Rightarrow^* v$ in G that does not include x or y only if both u and v belong to G_1 or both belong to G_2. We prove that no embedded K_4 exists in G by showing that there is no way to distribute four vertices x_1, x_2, x_3, and x_4 between G_1 and G_2 so that there are six pairwise disjoint paths $x_i \Rightarrow^* x_j$, $1 \le i < j \le 4$, in G.

Clearly not all four vertices can be in either G_1 (or G_2) or by the definition of merging G_1 (or G_2) would contain K_4 as an embedded

subgraph. Let $x_1 \notin G_1$ and x_2, x_3, and x_4 be in G_1; then all three paths $x_1 \Rightarrow^* x_2$, $x_1 \Rightarrow^* x_3$, and $x_1 \Rightarrow^* x_4$ contain x or y or both so they cannot be pairwise disjoint. Similarly if two vertices belong to G_1 and two do not, there would be four of the paths that include x or y or both so once again they could not be disjoint. Because of the symmetry between G_1 and G_2 this covers all possible ways of distributing the four vertices between the two graphs and we conclude that G does not contain K_4 as an embedded subgraph.

We now prove that if N is not TTSP, then it contains K_4 as an embedded subgraph.

If N is not TTSP, at least one of its triconnected components is a triconnected graph with four or more vertices. Thus according to Lemma C.1 that component contains K_4 as an embedded subgraph and according to Lemma C.3 so will N. □

Lemma 4.1.

(a) Let G be a TTSP multidigraph. The TT network obtained by adding a return edge (joining the terminals of G) to the undirected version of G is a TTSP network.

(b) Let N be a TTSP network. The multidigraph obtained by assigning directions to the edges of N as described earlier and deleting the return edge is a TTSP multidigraph.

Proof.

(a) We use induction on the number of edges of G.

If G has one edge the proposition is obviously true since N(G) -- the TT network obtained from G by the operation described in the lemma -- is a double bond.

Assume now that the proposition is true for all TTSP multidigraphs with fewer than k edges and that G has k edges. The TTSP multidigraph G is formed by the two terminal series or two terminal parallel composition of two TTSP multidigraphs G_1 and G_2, each having at most $k-1$ edges so that $N(G_1)$ and $N(G_2)$ are TTSP networks by induction hypothesis. But clearly $N(G)$ has a decomposition whose components are $N(G_1)$ and $N(G_2)$ and whose core is either a triangle or a triple bond (depending on the operation used to construct G from G_1 and G_2), therefore by Lemma 3.7 $N(G)$ is a TTSP network.

(b) We use once again induction on the number of edges of N and the close relationship between two terminal series (parallel) compositions and decompositions whose cores are triangles (triple bonds).

If N has fewer than four edges it is either a double bond, a triple bond, or a triangle, and the proposition is clearly true since $M(N)$ -- the multidigraph produced by the process described in the lemma -- consists, respectively, of a single edge, two edges in parallel, or two edges in series.

If N has more than four edges, according to Lemma 3.7, it has a decomposition whose core N_0 is a triangle or a triple bond and whose components are TTSP networks N_1 and N_2. Each of the components has fewer edges than N so by induction hypothesis $M(N_1)$ and $M(N_2)$ are TTSP multidigraphs. But $M(N)$ can clearly be constructed by the two terminal series or two terminal parallel composition (depending on whether N_0 is a triangle or a triple bond) of $M(N_1)$ and $M(N_2)$, therefore we conclude that $M(N)$ is a TTSP multidigraph. □

Lemma 4.2. TTSP multidigraphs are acyclic.

Proof. We prove the proposition by induction, showing that the two terminal series or two terminal parallel composition of two acyclic multidigraphs is an acyclic multidigraph.

If G is the two terminal series composition of G_1 and G_2, there is no path $u \Rightarrow^* v$ in G such that $u \in G_2$ and $v \in G_1$ since the only common vertex to G_1 and G_2 is a source of G_2 and a sink of G_1. If a cycle exists in G that is not contained entirely in G_1 or G_2 such a path would also exist, so every cycle of G has to be contained entirely in G_1 or G_2 and therefore G has no cycles.

The same argument can be repeated when G is the two terminal parallel composition of G_1 and G_2 to complete the proof. □

Lemma 4.3. A multidigraph is TTSP if and only if it can be reduced to a single edge by an appropriate sequence of Series and Parallel reductions.

Proof. We prove that if G is a TTSP multidigraph it can be reduced to a single edge by induction on the number of edges of G.

If G has one edge the is obviously true; otherwise let G have k edges. By definition G can be constructed by the two terminal series or two terminal parallel composition of two TTSP digraphs G_1 and G_2, each one having at most $k-1$ edges. By induction hypothesis, there are appropriate sequences of series and parallel reductions that transform G_1 and G_2 independently into a single edge. Because a reduction that creates an edge (u,v) does not depend on what is outside (u,v) but only on what was between the two vertices, the two sequences of reductions that transform G_1 and G_2 into single edges can be applied to G to

transform it into two edges in series or two edges in parallel (depending on what operation constructs G out of G_1 and G_2). In either case one more reduction transforms G into a single edge.

We now prove the implication in the other direction using the same method.

If G has a single edge it obviously has to be a TTSP multidigraph. Otherwise let G have k edges and be reducible to a single edge. Consider the last step in the reduction of G: that will be a reduction of two edges e_1 and e_2, either in parallel or in series, to a single edge. Each of these two edges has clearly arisen by the reduction of two subgraphs of G, G_1 and G_2, to a single edge. Each of the subgraphs can have at most $k-1$ edges, so by induction hypothesis both are TTSP multidigraphs. Because series and parallel reductions do not create or destroy sources or sinks, the source and sink of G_1 are the endpoints of e_1 and the source and sink of G_2 are the endpoints of e_2. Thus G can be constructed by two terminal series (if e_1 and e_2 are in series) or two terminal parallel (if e_1 and e_2 are parallel) composition of G_1 and G_2 and is a TTSP multidigraph. □

Lemma 4.5. There is a branch-in vertex of G_E that is a successor of a branch-out vertex.

Proof. We start our proof by showing that G_E contains one branch-out vertex. (Remember that G_E is acyclic, has a single source and sink, and each of its vertices is a branch-in, a branch-out, or both.)

If the source of G_E has two distinct successors, we have found our branch-out vertex. Otherwise the source has a unique successor v. This

vertex cannot be a branch-in or G_E would contain a cycle so it must be a branch-out.

We now complete our argument by showing how from any non-empty set S of branch-out vertices of G_E either (i) one of the members of the set has a branch-in successor or (ii) we can find a larger set S' of branch-out vertices of G_E.

The set S is defined using S by: $S' = \{x \mid \exists y, y \in S \text{ and } (y,x) \in G_E\}$. If no element of S' is a branch-in vertex, $\|S'\| \geq 2\|S\|$ since each element of S has at least two successors.

Because the number of branch-out vertices of G_E is bound by the total number of vertices, the process of finding ever larger sets of these vertices cannot be repeated indefinitely in a finite graph and our porposition must be true. □

Lemma 5.1.

(i) MSP, GSP, and TSP digraphs are acyclic and contain no multiple edges.

(ii) MSP digraphs are minimal.

(iii) TSP digraphs are transitive.

(iv) The transitive closure of any MSP digraph (and therefore of any GSP digraph as well) is a TSP digraph.

(v) The transitive reduction of any TSP digraph is an MSP digraph.

Proof. All the propositions of the lemma can be stated as properties of the edges of the digraphs and then proved by induction showing that the operations that introduce new edges preserve the property. As an example we prove proposition (ii) by showing that no edge of an MSP digraph is redundant by induction on the number of vertices of the digraph.

If the MSP digraph contains a single vertex, the proposition is trivially true; otherwise let the proposition be true of all MSP digraphs with fewer than k vertices and let $G = \langle V,E \rangle$ be an MSP digraph with exactly k vertices constructed by minimal series or parallel composition of two MSP digraphs $G_1 = \langle V_1,E_1 \rangle$ and $G_2 = \langle V_2,E_2 \rangle$ each one having at most $k-1$ vertices.

If G is the parallel composition of G_1 and G_2 the proposition is true because no edge of G_1 or G_2 is redundant by induction hypothesis and every edge of G is an edge of G_1 or an edge of G_2 since no new edges are introduced in a parallel composition.

If G is the minimal series composition of G_1 and G_2, we can argue in the same manner for any edge of G that belongs to G_1 or to G_2, therefore we only have to show that the edges of $E-(E_1 \cup E_2)$ are not redundant. Let $e = (x,y)$ be one such edge. By definition, x is a sink of G_1 and y is a source of G_2. Assume that e is redundant, that is, there is a path $p: x \Rightarrow^* y$ in G that does not include e. Let (x,u) and (v,y) be the first and last edges on p; because x was a sink of G_1, u must belong to G_2, and because y was a source of G_2, v must belong to G_1. We have therefore found a path $u \Rightarrow^* v$ in G in which $u \in G_2$ and $v \in G_1$; but this is absurd since no edge of G that leaves a vertex of G_2 enters a vertex of G_1, therefore we must conclude that p does not exist and that e is not redundant. □

Lemma 5.2. Let G_1 and G_2 be two multidigraphs having a single source and a single sink. Let G_{TTS} and G_{TTP} stand respectively for the Two Terminal Series and Two Terminal Parallel compositions of G_1 and G_2, and let $L(G)$ indicate the line digraph of digraph G (see Appendix A for definition).

(i) $L(G_{TTS})$ is the minimal series composition of $L(G_1)$ and $L(G_2)$.

(ii) $L(G_{TTP})$ is the parallel composition of $L(G_1)$ and $L(G_2)$.

Proof. We prove proposition (i) by showing that for any two edges $e_1 \in G_1$ and $e_2 \in G_2$, there is an edge $(\ell(e_1), \ell(e_2))$ in $L(G_{TTS})$ if and only if $\ell(e_1)$ was a sink of $L(G_1)$ and $\ell(e_2)$ a source of $L(G_2)$.

The vertex $\ell(e_1)$ of $L(G_1)$ is a sink if and only if e_1 enters the only sink of G_1, because if $e_1 = (x,y)$ and y is not a sink, there must be another edge $e = (y,z)$ consecutive with e_1 in G_1 and there would be an edge $(\ell(e_1), \ell(e_2))$ in $L(G_1)$. By a similar argument we can prove that $\ell(e_2)$ will be a source of $L(G_2)$ if and only if e_2 leaves the only source of G_2.

By definition there will be an edge $(\ell(e_1), \ell(e_2))$ in $L(G_{TTS})$ if and only if the edges e_1 and e_2 are consecutive in G_{TTS}. Because $e_1 \in G_1$ and $e_2 \in G_2$ the only case in which they would be consecutive is when e_1 enters the sink of G_1 and e_2 leaves the source of G_2, and in that case $\ell(e_1)$ is a sink of $L(G_1)$ and $\ell(e_2)$ a source of $L(G_2)$ and proposition (i) is proved.

Proposition (ii) can be proved by a similar but simpler argument that we omit. □

Lemma 5.3. Let G be a multidigraph with one source and one sink. G is TTSP if and only if $L(G)$ is an MSP digraph.

Proof. Consider the following one-to-one relationship between the members of the two classes of digraphs:

-- the TTSP multidigraph consisting of a single edge corresponds to the MSP digraph with a single vertex; and

-- the TTSP multidigraph resulting from the two terminal series (two terminal parallel) composition of the TTSP multidigraphs G_1 and G_2, corresponds to the MSP digraph constructed by minimal series (parallel) composition of the MSP digraphs that correspond to G_1 and G_2.

The relationship of Lemma 5.2 can be used to prove by a straightforward inductive argument that for any TTSP multidigraph G, its corresponding MSP digraph G' is such that $L(G) = G'$. Thus G will be TTSP if and only if $L(G)$ is MSP. □

Lemma 5.6.

(i) CBC digraphs are minimal.

(ii) The bipartite components of a CBC digraph are unique.

(iii) Any MSP digraph is CBC.

Proof. (i) Let G be a CBC digraph. Assume that edge (u,v) is redundant, that is, there is a path $p: u \Rightarrow^* v$ in G that does not include (u,v), and let B_i be the bipartite component of G that includes (u,v). Let (u,x) and (y,v) be the first and last edges of p respectively (p has to contain at least two edges since G is a digraph). Because $h(y) = t(x) = B_i$, there must be an edge (y,x) in G. Now, if $x = y$ that edge would be a loop and G would not be a digraph, and if $x \neq y$, G would contain a cycle (formed by the section of p between x and y and the edge (y,x)) and it wouldn't be CBC either according to our definition. We therefore must conclude that G does not contain redundant edges.

(ii) Let G be a CBC digraph having two distinct sets of bipartite components: $S = \{B_1, B_2, \ldots, B_k\}$ and $S' = \{B'_1, B'_2, \ldots, B'_\ell\}$. We will show that the two sets are really identical by proving that for any component B_i of S, there is a component B'_j of S' such that $B_i = B'_j$.

Let (u,v) be an edge of B_i. The head of B_i contains exactly all the predecessors of v and its tail contains exactly all the successors of u. Now let B'_j be the component of S' including (u,v). Clearly the head and tail of B'_j must be identical to those of B_i and since both B_i and B'_j are complete bipartite digraphs, $B_i = B'_j$.

(iii) Let G be an MSP digraph. Each edge of G was introduced by a minimal series composition during the construction of G using the rules of Definition 5.1 and the minimal series composition of G_1 and G_2 introduces a set of edges that form a complete bipartite digraph whose head is the set of sinks of G_1 and whose tail is the set of sources of G_2. Let these complete bipartite digraphs $B_1, B_2, \ldots, B_k$, be the components of G. Clearly each edge of G belongs to exactly one component. Furthermore, each vertex u of G, that is not a sink, belongs to the head of at least one subgraph B_i, and it could not belong to the head of more than one since after the composition that creates B_i, u would not be a source anymore so no new edges leaving it could be introduced ever after. The same reasoning proves that each vertex of G that is not a source belongs to the tail of exactly one of the components. This is enough to guarantee that G satisfies Definition 5.3 and is therefore CBC. □

Lemma 5.7. $L(L^{-1}(G)) = G$ for any CBC digraph.

Proof. By definition, for every vertex x of G that is not a source or a sink, there is an edge $e_x = (t(x),h(x))$ in $L^{-1}(G)$, and for each edge e_x of $L^{-1}(G)$ there will be a vertex $\ell(e_x)$ in $L(L^{-1}(G))$. We prove that G and $L(L^{-1}(G))$ are isomorphic by proving that there is an edge (u,v) of G if and only if there is an edge $(\ell(e_u),\ell(e_v))$ in $L(L^{-1}(G))$.

Let (u,v) be an edge of G and let B_i be the bipartite component of G that includes (u,v). Clearly, $h(u) = t(v) = B_i$ and therefore the edges $e_u = (t(u),h(u))$ and $e_v = (t(v),h(v))$ are consecutive in $L^{-1}(G)$. As a consequence of these edges being consecutive, there will be an edge $(\ell(e_u),\ell(e_v))$ in $L(L^{-1}(G))$.

If there is no edge (u,v) in G, $h(u) \neq t(v)$, the edges e_u and e_v will not be consecutive in $L^{-1}(G)$ and there will be no edge $(\ell(e_u),\ell(e_v))$ in $L(L^{-1}(G))$. □

Lemma 5.8. Let G be an acyclic digraph and (u,v) a redundant edge of G.

$$M_G(u) < J_G((u,v)) .$$

Proof. If (u,v) is redundant in G, there must be a path $p: u \Rightarrow^* v$ in G that does not include (u,v). Because G is a digraph, the path has to include at least two edges, so let (u,x) be the first edge of that path. Because the values of L_G must increase along any path in G, and there is a path $x \Rightarrow^* v$, $L_G(x) < L_G(v)$. Therefore by definition $J_G((u,x)) < J_G((u,v))$ and the proposition must be true since $M_G(u) \leq J_G((u,x))$. □

Lemma 5.9. Let G be an MSP digraph. For any edge (u,v) of G,

$$M_G(u) = J_G((u,v)) .$$

Proof. We prove the proposition by induction on the number of vertices of G.

If G has one vertex, the proposition is trivially true; otherwise let the proposition hold for any MSP digraph with fewer than k vertices, and let G have exactly k vertices and be the minimal series or parallel composition of G_1 and G_2, both of which are MSP digraphs with at most $k-1$ vertices.

Let G be the parallel composition of G_1 and G_2. Any vertex or edge of G belongs to either G_1 or G_2 so let $(u,v) \in G_1$. By induction hypothesis $J_{G_1}((u,v)) = M_{G_1}(u)$ and because no edges are introduced in the composition $L_G(x) = L_{G_1}(x)$ for any vertex $x \in G_1$. Because the values of J_G and M_G are defined in terms of the values of L_G and these values are identical to those of L_{G_1} on the vertices of G_1, we conclude that $J_G((u,v)) = M_G(u)$ when $(u,v) \in G_1$.

The same argument can be used if $(u,v) \in G_2$, so if G is the parallel composition of G_1 and G_2 the proposition is true.

Let G be the minimal series composition of G_1 and G_2. We consider three cases: (i) $(u,v) \in G_1$, (ii) $(u,v) \in G_2$, and (iii) $(u,v) \in G-(G_1 \cup G_2)$.

For any vertex $y \in G_1$, $L_G(y) = L_{G_1}(y)$ so in case (i) the argument employed when G was the parallel composition of G_1 and G_2 can be repeated to prove the proposition.

Let now $(u,v) \in G_2$ and let q be the length of the largest path of G_1. This path has to end in a sink of G_1 and therefore, by definition of minimal series composition, for any vertex $z \in G_2$, $L_G(z) = L_{G_2}(z)+q+1$. Because J_G is defined by the difference of two values of L_G, for any edge $(a,b) \in G_2$, $J_G((a,b)) = J_{G_2}((a,b))$. Furthermore, since M_G is defined in terms of J_G, for any vertex $z \in G_2$, $M_G(z) = M_{G_2}(z)$. Since we know that $J_{G_2}((u,v)) = M_{G_2}(u)$ by induction hypothesis, we conclude that when $(u,v) \in G_2$, $J_G((u,v)) = M_G(u)$ and the proposition is true.

Finally, let $(u,v) \in G-(G_1 \cup G_2)$. In this case vertex u is a sink of G_1 so every edge leaving u enters a source of G_2. But we know that for any source w of G_2, $L_G(w) = q+1$ so for any edge e leaving u, $J_G(e) = q+1 - L_G(u)$ and therefore for all of them $M_G(u) = J_G(e)$. Thus the proposition is true in case (iii) as well and we conclude that it holds for all MSP digraphs. □

Lemma 5.11. Let G be a GSP digraph. G does not contain N as an implicit subgraph.

Proof. Let G_T be the transitive closure of G. Clearly G_T will be a TSP digraph and if G contains an implicit N subgraph, G_T would contain an induced N subgraph. We will prove that no TSP digraph contains an induced N subgraph -- which clearly implies that the lemma is true -- by induction on the number of vertices of the TSP digraph G_T.

If G_T has fewer than four vertices, the lemma is obviously true; otherwise let the lemma hold for all TSP digraphs having fewer than k vertices, and let G_T have exactly k vertices. The digraph G_T has

to be the series or parallel composition of two TSP digraphs G_1 and G_2, each having at most k-1 vertices. By induction hypothesis neither G_1 nor G_2 contains an induced N subgraph.

If G_T is the parallel composition of G_1 and G_2, the proposition is true because no edge joins a vertex of G_1 to a vertex of G_2 and thus every connected subgraph of G_T has to be a subgraph of G_1 or a subgraph of G_2.

Let then G_T be the series composition of G_1 and G_2. By definition, there will be an edge joining each vertex of G_1 to each vertex of G_2. Therefore every induced subgraph S of G_T will contain as a subgraph a complete bipartite digraph with head $S \cap G_1$ and tail $S \cap G_2$. It is trivial to test that the vertices of the N digraph cannot be split in such a way between G_1 and G_2, and therefore we must conclude that G_T does not contain an induced N subgraph. □

Lemma 5.12. Let $(u,v) \in E_T$. Either (u,v) is redundant in G or there are edges (u,x) and (y,v) in G such that $J_G((y,v)) = 1$ and $M_G(u) = J_G((u,x))$ and therefore x, y, u, and v are the four vertices of an implicit N subgraph of G.

Proof. The vertex x must exist because of the way in which the edges of E_T were determined. To show that vertex y must exist, let p be the longest path of G that starts at a source and ends at v; clearly (u,v) cannot be on that path or it would not have been deleted, so let y be the last vertex on p before v. By definition $L_G(v) = L_G(y)+1$.

Because (u,v) was in E_T, $L_G(v) > L_G(x)$ and therefore $L_G(y) \geq L_G(x)$. The values of L_G must increase along any path of G, therefore there

cannot be any path $y \rightarrow^* x$ in G, and if there is a path $x \rightarrow^* y$ the edge (u,v) would be redundant since (u,x) and (y,v) are edges of G.

Therefore either (u,v) is redundant or the vertices x, y, u and v form an implicit N subgraph of G. □

Lemma 6.1. Let H be a hammock such that $N(H)$ is biconnected and let S be a non-trivial subhammock of H. Either S includes every edge of H except the return edge or the entry and exit vertices of S, are a separation pair of $N(H)$.

Proof. The edges of $N(H)$ can be partitioned into two sets, one including those edges that correspond to edges of S and the other including the rest. If S does not include all edges of H except its return edge, there must be at least two edges in each set since S is non-trivial. Because S has just two boundary vertices, they will be the only vertices incident to edges of both sets and must therefore be a separation pair of $N(H)$. □

Lemma 6.2. Let H be a proper program. $N(H)$ is biconnected.

Proof. Let v be an articulation point of $N(H)$ and let H_1 and H_2 be the subgraphs of $N(H)$ separated by v. By definition of proper program, the vertex v can have at most degree three, so there must be one of the subgraphs, say H_2, that includes only one edge e incident to v. Clearly, e is a bridge of $N(H)$ separating the subgraphs H_1 and $H_2' = H_2 - \{v\}$. Because α and ω are adjacent in $N(H)$ they must belong to the same subgraph, H_1 or H_2'. In either case, no matter

what direction e has in H, there could not be paths $\alpha \rightarrow^* x$ and $x \rightarrow^* \omega$ in H for a vertex that belongs to the subgraph that does not include α and ω. Since we assumed that H was a proper program and therefore a hammock, v cannot be an articulation point of N(H) and the lemma must therefore be true. □

Lemma 6.3. Let H be a proper program with start vertex α and finish vertex ω, and let S be a subhammock of H. The digraph H' obtained by replacing S by a single edge from its entry to its exit is a proper program with start vertex α and finish vertex ω.

Proof. There are two facts to be proved: that H' is a hammock, and that all its vertices are function, predicate, or collect nodes. The first fact follows immediately from property (C2) (given in Section 6.2) of the definitions of entry and exit that we are employing. The second fact can be proved as follows: the entry and exit of S must each have at least one edge of S incident to them, so the replacement does not increase the total number of edges incident to either. Now, if at most three edges are incident to a vertex v of a hammock, it must be a function, predicate, or collect node, or otherwise there would be no way of reaching v or to exit v. Therefore H' is a proper program. □

Lemma 6.4. Let H be a proper program and let S be a subgraph of N(H) that does not include the return edge. The subgraph S can be reduced to a single edge by one series, parallel, or triconnected reduction (as defined in Chapter 3) if and only if the subgraph S' of H containing all the vertices and edges of S is a prime subhammock.

Proof. This lemma is true only for proper programs because it can only be proved using the fact that the vertices of proper programs have total degree two or three.

We start by proving that any boundary vertex v of a subgraph S_1 of a proper program H_1 is either an entry or an exit of S_1.

Because at most these edges of H_1 are incident to v, either S_1 or H_1-S_1 must contain exactly one of these edges. In either case according to our definitions v is either an entry or an exit.

We prove the lemma now by proving first that if S can be eliminated by a single reduction in $N(H)$, S' is a prime subprogram of H and then that if S' is a prime subprogram of H, S can be eliminated by a single reduction in $N(H)$.

A subgraph of $N(H)$ that can be eliminated by a single reduction has to have exactly two boundary vertices and be a double bound, a triconnected graph with at least four vertices or consist of two edges in series. The two boundary vertices of S will be entries or exits of S' by the argument given earlier. Furthermore, they must be an entry - exit pair or otherwise for some vertex $x \in S'$ there would be no path $\alpha \Rightarrow^* x$ or no path $x \Rightarrow^* \omega$ in H and H would not be a hammock. Thus S' must be a non-trivial subhammock of H. But S does not include any separation pairs other than its boundary vertices, so according to Lemma 6.1, no proper subgraph of S' is a subhammock and S' must be a prime subhammock of H.

The implication in the other direction can be proved by a very similar argument. If S' is a prime subhammock of H, it must have exactly two boundary vertices, and therefore so will S. Because every boundary

vertex of a subgraph of H would be an entry or an exit of the subgraph, and S' does not properly contain any subhammock, no proper subgraph of S' can have exactly two boundary vertices and contain at least two edges. Thus S will not contain a separation pair and therefore has to be either a double bond, a triconnected graph, or consist of two edges in series and in all cases it can be eliminated by a single reduction. □

Lemma 6.5. Let H be a structured program. N(H) is biconnected.

Proof. We prove the proposition by induction on the number of "expansion" operations needed to construct H from the pseudo-hammock of Figure 6.16(a).

The proposition is obviously true if H is one of the hammocks of Figure 6.16(b) which are the only structured programs that can be obtained by a single expansion operation. If H is not one of these hammocks, let the propostion be true for all structured programs constructed by fewer than k expansions and let H be constructed by exactly k such operations. In this case there must be a structured program H' from which H can be generated by one expansion operation, and by induction hypothesis N(H') must be biconnected. The vertices of H introduced on the last expansion cannot be articulation points of N(H) because they are not articulation points of the subgraph introduced in the operation and this subgraph has two boundary vertices. Any other vertex of H cannot be an articulation point of N(H) either because any such vertex would also be an articulation point of N(H') and we know that N(H') is biconnected. We therefore conclude that no vertex of N(H) is an articulation point and that N(H) is a biconnected subgraph. □

Lemma 6.6. Let H be a structured program. N(H) can be reduced to a double bond by a sequence of series and parallel reductions that do not involve the return edge.

Proof. We use once again induction of the number of operations needed to construct H from the pseudo-hammock of Figure 6.16(a).

The proposition is obviously true of the structured programs of Figure 6.16(b) which are the only ones that can be generated by one expansion operation. Let the proposition be true now for all structured programs generated by fewer than k expansions and let H be constructed by exactly k such operations. In that case there must be a structured program H' from which H can be generated by a single expansion, and by induction hypothesis N(H') can be reduced to a double bond as described in the lemma. Now the subgraph of N(H) introduced by the last expansion operation can be reduced to a single edge by series and parallel reductions, (since that subgraph must be one of the structured programs of Figure 6.16(b)), and then either its entry or exit can be eliminated by series reduction. In this manner N(H) has been converted into N(H') by series and parallel reductions and since N(H') can be converted into a double bond by a sequence of these reductions, the same is true of N(H) and the proposition is proved. □